PROCEEDINGS OF THE TWELFTH ANNUAL NATIONAL CONFERENCE ON Ada TECHNOLOGY

MARCH 21-24, 1994

Sponsored By:
ANCOST, INC.

With Participation By:
United States Army
United States Navy
United States Air Force
United States Marine Corps
Ada Joint Program Office
Defense Information Systems Agency
Federal Aviation Administration
National Aeronautics & Space Administration

Academic Host:
Norfolk State University

The Ada Joint Program Office
of the Defense Information Systems Agency
has graciously provided funding for
the printing of these Proceedings.

PROCEEDINGS OF THE TWELFTH ANNUAL NATIONAL CONFERENCE ON Ada TECHNOLOGY

Sponsored By:
ANCOST, INC.

With Participation By:
United States Army
United States Navy
United States Air Force
United States Marine Corps
Ada Joint Program Office
Defense Information Systems Agency
Federal Aviation Administration
National Aeronautics & Space Administration

Academic Host:
Norfolk State University

Williamsburg Hilton--Williamsburg, VA

March 21-24, 1994

The Papers in this volume were printed directly from unedited reproducible copies prepared by the authors. Responsibility for contents rests upon the author, and not the conference committee or its members. After the conference, all publication rights of each paper are reserved by their authors, and requests for republication of a paper should be addressed to the appropriate author. Abstracting is permitted, and it would be appreciated if the conference is credited when abstracts or papers are republished. Requests for individual copies of papers should be addressed to the authors.

Approved for Public Release: Distribution Unlimited

TWELFTH ANNUAL NATIONAL CONFERENCE ON Ada TECHNOLOGY CONFERENCE COMMITTEE 1993-1994

President:
DR. JAMES HOOPER, Marshall University

Treasurer:
MS. SUSAN MARKEL, TRW

Secretary:
DR. MURRAY KIRCH, Stockton College of New Jersey

President-Elect:
MS. CHRISTINE L. BRAUN, GTE Federal Systems

Immediate Past President:
MR. STEVE LAZEROWICH, Virtual Software Factory, Inc.

Academic Host Chair:
DR. GEORGE HARRISON, Norfolk State University

Academic Outreach:
DR. MARTIN BARRETT, Penn. State Univ. at Harrisburg

Budget Committee Chair:
MR. MICHAEL SAPENTER, Telos Federal Systems

Policies, Procedures, & By-Laws Chair:
MS. DEE GRAUMANN, GDE Systems. Inc.

Panels Chair:
MR. CURRIE COLKET, U.S. Navy

Promotion Chair:
MS. JUDITH M. GILES, Intermetrics, Inc.

Technical Program Chairs:
MR. DANIEL HOCKING, Army Research Laboratory

DR. AKHTAR LODGHER, Marshall University

Tutorial Chair:
DR. C. RONALD GREEN, U.S. Army, CSSD

MR. MIGUEL CARRIO JR., MTM Engineering

DR. EMMANUEL OMOJOKUN, Virginia State University

MS. LAURA VEITH, Rational

ADVISORY MEMBERS...

MR. CURRIE COLKET,
U.S. Navy
CAPT. DAVID COOK,
U.S. Air Force Academy
MAJ. GERALD DEPASQUALE,
U.S. Marine Corps
MR. JEFFERY HERMAN,
U.S. Army CECOM, SED
MR. DANIEL HOCKING,
Army Research Laboratory
MR. HUET LANDRY,
DISA Center for Standards
MR. DONALD J. REIFER, AJPO
MR. E.V. (ZEKE) SALTER,
DISA
MR. CARRINGTON STEWART,
NASA
Conference Director:
MS. MARJORIE RISINGER, CMP,
Rosenberg & Risinger

The Annual National Conference on Software Technology Inc., has received the 1993 Technical Excellence Award from the National Technical Achiever Association. The National Technical Achiever of the Year Awards Program recognizes technical professionals of color for their contributions to the technical community and the community at large. The mission of the ANCOST is to actively promote the development of well-qualified software engineering professionals at HBCUs and other institutions, by conducting an annual conference and related activities that encourage ongoing interaction among academia, government, and industry.

Eleventh Annual National Ada Conference Committee

PROCEEDINGS
TWELFTH ANNUAL NATIONAL CONFERENCE ON Ada TECHNOLOGY

Bound--Available at Fort Monmouth

2nd Annual National Conference on Ada Technology, 1984--(N/A)
3rd Annual National Conference on Ada Technology, 1985--$10.00
4th Annual National Conference on Ada Technology, 1986--(N/A)
5th Annual National Conference on Ada Technology, 1987--(N/A)
6th Annual National Conference on Ada Technology, 1988--$20.00
7th Annual National Conference on Ada Technology, 1989--$20.00
8th Annual National Conference on Ada Technology, 1990--$25.00
9th Annual National Conference on Ada Technology, 1991--$25.00
10th Annual National Conference on Ada Technology, 1992--$25.00
11th Annual National Conference on Ada Technology, 1993--$25.00
12th Annual National Conference on Ada Technology, 1994--$25.00

Extra Copies: 1-3 $25.00 each; next 7 $20 each; 11 & more $15 each. Make check or bank draft payable in U.S. dollars to ANCOST and forward requests to:

Annual National Conference on Ada Technology
Rosenberg & Risinger
11287 W. Washington Blvd.
Culver City, CA 90230-4615

Telephone inquiries may be directed to Rosenberg & Risinger at 310/397-6338.

Photocopies--Available at Department of Commerce. Information on prices and shipping charges should be requested from:

U.S. Department of Commerce
National Technical Information Service
Springfield, VA 22151
USA

Include title, year, and AD number

2nd Annual National Conference on Ada Technology, 1984-ADA 142403
3rd Annual National Conference on Ada Technology, 1985-ADA 164338
4th Annual National Conference on Ada Technology, 1986-ADA 167802
5th Annual National Conference on Ada Technology, 1987-ADA 178690
6th Annual National Conference on Ada Technology, 1988-ADA 190936
7th Annual National Conference on Ada Technology, 1989-ADA 217979
8th Annual National Conference on Ada Technology, 1990-ADA 219777
9th Annual National Conference on Ada Technology, 1991-ADA 233469
10th Annual National Conference on Ada Technology, 1992-ADA 248007
11th Annual National Conference on Ada Technology, 1993-ADA 262517

Technical Program . . .

Tuesday, March 22, 1994

Opening Session: 8:30am - 10:00am
Keynote Address:
RADM Scott L. Sears
Commander, Naval Undersea Warfare Center

Dual Use Panel: 10:30 - 12 Noon

Moderator: David Basel, Deputy Director of AJPO, DISA, Center for Information Management 288

Luncheon: 12 Noon - 1:30pm
Speaker:
Dr. Arthur Hersch
President and CEO
Software Productivity Consortium

Invited Papers - OOP/Groupware: 2:00pm - 3:30pm
Chairman: Daniel E. Hocking

Speakers: Dr. C. Thomas Wu 252
Dr. Jay Nunamaker 257

Software Engineering: 2:00pm - 3:30pm
Chairman: Carrington Stewart

1.1 Comparison of Design Diversity Techniques for Fault Tolerant Ada Software Development - D.M. Coleman and R. Leach, Howard University, Washington, DC 6

1.2 An Overview of the Ada Semantic Interface Specification (ASIS)- W. Fay and V. Ohnjec, CAE Link Space Systems, Houston, TX 16

1.3 Ada's Role in Preventing Data Corruption: Theory Vs. Reality in a large Avionics Software Project - M. Jenkins, Honeywell, Phoenix, AZ 20

Metrics: 4:00pm - 5:30pm
Chairman: Christine Braun

2.1 The Role of the Ada Compiler Evaluation System in Selection and Use of a Compilation and Execution Environment - P. Brashear, CTA Incorporated, Dayton, OH 29

2.2 Implementation of Six Object-Oriented Metrics - W. Fay and V. Ohnjec, CAE Link Systems, Houston, TX 36

2.3 Data from an Ada Laboratory - A. Ahmed, King Saud University, Riyadh, Saudi Arabia .. 51

Ada 9X Panel: 4:00pm - 5:30pm

Moderator: Christine Anderson, Manager Ada9X, US Air Force 293

Vendor Hospitality Suites: 7:00pm - 10:00pm

Wednesday, March 23, 1994

Environment Assessment Methodology Panel: 8:30am - 11:45am

Moderator: Richard Petersen, Capt., USN, Deputy Director, Information Processing Directorate, DISA Center for Standards 298

Ada Language Issues: 8:30am - 10:00am
Chairman: Currie Colket

3.1 Ada Integration for an MS-Windows Tool Framework - P. Pukite, DAINA, Columbia Heights, MN 63

3.2 Software Reengineering Assesment Handbook - J. Clark and B. Stevens, COMPTEK Federal Systems, Virginia Beach, VA, J. Donald, Air Force Cost Analysis Agency, Arlington, VA, S. Stukes, Management Consulting & Research, Oxnard, CA 69

3.3 Application Specific Languages and Ada - C. Braun, GTE Federal Systems, Chantilly, VA 77

Education: The Undergraduate Student and Ada: 8:30am - 10:00am
Chairman: Dr. Akhtar Lodgher

4.1 Ada, a Software Engineering Tool, in Introductory Computer Science Programming Courses at Sacred Heart University: Mutual Benefits - S. H. Adams, Sacred Heart University, Fairfield, CT 84

4.2 Making Ada "Exciting and Fun" for Undergraduates - A. Lodgher and H. Al-Haddad, Marshall University, Huntington, WV 89

4.3 Teaching the Concept of Reusable Software Components with Ada - A. Al-Dehlaan and J. C. Agrawal, King Saud University, Riyadh, Saudi Arabia 97

Object Oriented Development: 10:15am - 11:45am
Chairman: Susan Markel

5.1 Cognitive Analysis, Design and Programming Toward Generalization of the Object Oriented Paradigm and the Reuse and Conversion of Legacy Code in Ada - J. Scandura, Univ. of PA 103

5.2 Experience in Testing Object-Oriented Ada Software - A. Graziano, IIT Research Institute, Annapolis, MD 108

5.3 A Systematic Object Identification Technique in Object-Oriented Paradigm - J. Monroe, NC A&T State University, Greensboro, NC 116

Education: Competition: 10:15am - 11:45am
Chairman: Dr. Martin Barrett

6.1 Ada Versus Pascal for Software Engineering Education and Practice - A. Al-Dehlaan and J. C. Agrawal, King Saud University, Riyadh, Saudi Arabia 124

6.2 Ada versus Modula-2: A Plan for an Empirical Comparison of their Effectiveness in Introductory Programming Courses - H.K. Dai and K.E. Flannery, University of North Dakota, Grand Forks, ND 129

6.3 A Note on Teaching Abstract Data Types Through Ada and C++ - K. H. Chung and H. C. Chu, National Defense Management College, Taipei, Taiwan 134

Keynote Address:
LTG Alonzo E. Short, Jr.
Director, Defense Information Systems Agency

HBCU Net Panel: 2:00pm - 3:30pm
Moderator: Maj. Shelton Lee, Jr. 300

Education: Competition of the Languages: 2:00pm - 3:30pm
Chairman: Dr. George Harrison

7.1 A Year-long Sequence in Software Engineering to use the Language Ada in Undergraduate Curriculum.- W. McArthur and R. Ruth 142

7.2 A Comprehensive Software Engineering Educational Experience - G. C. Harrison, Norfolk State University, Norfolk, VA 148

7.3 An Undergraduate Software Engineering Course Sequence using Ada - R. Y. Lee and J. P. Kelsh, Central Michigan University, Mount Pleasant, MI 154

Education: The Undergraduate Student and Software: 4:00pm - 5:30pm
Chairman: Dr. Murray Kirch

8.1 Ada: The Leitmotif of Software Engineering Education - J. K. Blundell, University of Missouri-Kansas City, Kansas City, MO 162

8.2 Ada Beyond Implementation - S. Smith, Converse College, Spartanburg, SC, and D. Gotterbarn and R. Riser, East Tennessee State University, Johnson City, TN 167

8.3 An Ada/Software Engineering Migration Training Plan - J.W. Hooper, A. Lodgher, H. Al-Haddad, Marshall University, Huntington, WV 184

DOD OSE Guidance Panel: 4:00pm - 5:30pm
Moderator: Huet Landry, DISA 307

Thursday, March 24, 1994

Opening Session: 8:30am - 9:15am
Keynote Address:
The Honorable Emmett Paige, Jr.
Assistant Secreatry of Defense for Command, Control, Communications and Intelligence

AdaSAGE Panel: 9:30am - 11:00am
Moderator: Joan McGarrity, Naval Computer and Telecommunications Command 323

Methods: 9:30am - 11:00am
Chairman: Michael Sapenter

9.1 The Product-Driven Development of a Real-Time Ada Executive for the Space Shuttle Cockpit Upgrade - C.A. Meyer, D. Soto, G. Barber, Intermetrics, Inc., Houston, TX 195

9.2 Software Logistics: Knocking at the Army's Door - J. Potoczniak, CECOM Research, Fort Monmouth, NJ 206

9.3 Software Process Definition for an Ada Program with Significant Software Reuse - T. Shetler, TRW Systems Division, Fairfax, VA 216

Ada Resources Panel: 11:15am - 12:45pm
Moderator: Susan Carlson, AIC 332

Reuse: 11:15am - 12:45pm
Chairman: Dr. Ronald Green

10.1 Identifying Reuse Opportunities Within and Across Organizational Boundaries - R.L. Owens & M.H. Riggs, SofTech Inc., Alexandria, VA. 226

10.2 Developing Reusable Objects in Ada: A First Time Experience W. Schoen, IIT Research Institute, Annapolis, MD. 240

10.3 Integrating Ada into Reusable Software Engineering - H. Yu and J. Monroe, NC A&T State University, Greensboro, NC. 248

Luncheon: 12 :45pm - 2:30pm
Speaker:
Ralph E. Crafts
President, Ada Software Alliance

NOTES

SCOTT L. SEARS

Rear Admiral, USN Commander

Rear Admiral Scott L. Sears assumed command of the Naval Undersea Warfare Center (NUWC) in August 1992.

He reported to NUWC after serving as Program Manager of the AN/BSY-2 Program Office, a position he held from March 1989. As Program Manager, he was responsible for developing the combat system for the next generation attack submarine SEAWOLF.

A native of Cleburne, TX, Rear Admiral Sears graduated from the United States Naval Academy with distinction in June 1966. Further academic achievements include earning a master of science degree in electrical engineering and an electrical engineer degree (professional degree) from the Massachusetts Institute of Technology under the Burke Scholar Program. He has also graduated from Duke University's School of Business Advanced Management Program.

Duty on submarines USS BARBEL (SS 580), USS GUDGEON (SS567), USS SPADEFISH (SSN 668), USS FLYING FISH (SSN 673), and USS HENRY L. STIMSON (SSBN 655) (Blue) prepared Rear Admiral Sears for command of USS ALBUQUERQUE (SSN 706) from October 1983 to December 1986. Under his command, USS ALBUQUERQUE won the 1984 CINCLANTFLT Golden Anchor Award for retention excellence, the 1986 Submarine Squadron Two ASW "A" for excellence in antisubmarine warfare and a Meritorious Unit Commendation.

Following command, Rear Admiral Sears served as Deputy Commander of Training and Operations at Submarine Squadron Seventeen and as Head, Tactical Weapons Branch of the Attack Submarine Division of the staff of the Chief of Naval Operations. He was selected for Flag Rank as a Material Professional in January 1991.

Decorations include two awards of the Legion of Merit, three Meritorious Service Medals, two Navy Commendation Medals, and two Navy Achievement Medals.

Rear Admiral Sears is married to the former Barbara Rost. They have two children, Paul and Kathy.

ARTHUR I. HERSH

President and Chief Officer
Software Productivity Consortium

Arthur I. Hersh is president and chief executive officer of the Software Productivity Consortium. He joined the Consortium in December, 1989, following a 32-year career with GTE Government Systems Corporation, in Waltham, Massachusetts. He is also the President of the Virginia Center of Excellence for Software Reuse and Technology Transfer(VCOE).

Mr. Hersh directs all activities of the Consortium. Under his guidance, the Consortium Develops innovative processes, methods, tools, and services that help its members to significantly improve their software engineering practices. Consortium products are currently being used by its members and government customers on some of the nations largest and most critical programs.

As the principal point of contact with the Consortium's member companies, Mr. Hersh ensures that the technologies, services, and transfer mechanisms developed by the Consortium meet the needs of all members. He is responsible for setting the strategic and tactical goals of all Consortium divisions and projects, and manages the fiscal and human resources of the Consortium to most effectively meet these goals.

At GTE, Mr. Hersh held key management responsibility for command, control, communications and intelligence(C'I) systems. He last served as Vice President, Programs and Engineering. He held numerous other executive management positions with GTE, including Group Vice President and General Manager of the Command, Control and Communications Systems sector, and Vice President & General Manager, Communications Systems Division, Mr. Hersh has in-depth program management experience, with a special emphasis in large-scale, software-intensive telecommunications systems.

Mr. Hersh holds a B.S.E.E. from the University of Pittsburgh, an M.S.E.E. from the University of Southern California and an E.E. from the Massachusetts Institute of Technology. He is a member of the Army Science Board, the Industrial Advisory Board for the Northeastern University School of Engineering and the IEEE Software Industrial Advisory Board. He is also a member of the steering committee for the Center for Strategic and International Studies' current study, "National Benefits from National Laboratories." He is on the Board of Directors, Government Division, of the Electronic Industries Association(EIA). Mr. Hersh is a registered engineer and an active participant in many professional associations, including AFCEA, EIA, AEA, NSIA, and IEEE.

Mr. Hersh and his wife reside in Potomac, Maryland.

LIEUTENANT GENERAL ALONZO E. SHORT, JR.

Director, Defense Information Systems Agency/Manager, National

Lieutenant General Alonzo E. Short, Jr., was born in Greenville, North Carolina, on 27 January 1939. He graduated from Virginia State College with a bachelors of science degree in education and a masters degree in business management from the New York Institute of Technology. He holds honorary doctorates from Virginia State University and C.H. Mason University. His military education includes completion of the Signal Officer Basic and Advanced Courses, the Armed Forces Staff College, the Communications-Electronics Systems Engineer Course, and the Army War College.

Since entering the Army in June of 1962, General Short has held a variety of assignments. He was a platoon leader and staff officer at Fort Riley, Kansas. From 1965 to 1967, he was a staff officer, company commander, and an executive officer of a signal battalion in Europe. In 1967, he was assigned as a staff planning and engineering officer in the Republic of Vietnam, followed by assignment to Okinawa, first at battalion S-3, the executive officer and finally battalion commander in the Strategic Communications Command-Okinawa Signal Group. He also served a second tour in Vietnam in 1972-73 as an advisor.

In 1975, General Short was assigned as a staff officer in the Defense Communications Agency later becoming a battalion commander in the 101st Airborne Division at Fort Campbell, Kentucky.

In 1979, he began a tour as a staff planner with the Army Communications Command at Fort Huachuca. He then served as Commander of the 3rd Signal Brigade at Fort Hood, Texas.

The General was then assigned as the Deputy Commander of the Army Electronics Research and Development Command (ERADCOM), Adelphi, Maryland, from July 1984 to October 1984.

General Short was promoted to Brigadier General concurrent with assuming command of the Information Systems Management Activity while at the same time taking over as Program Manager for Army Information Systems, at Fort Monmouth, New Jersey. In July 1986, he became Deputy Commander of the Information Systems Engineering Command (ISEC), and Commander in September 1987.

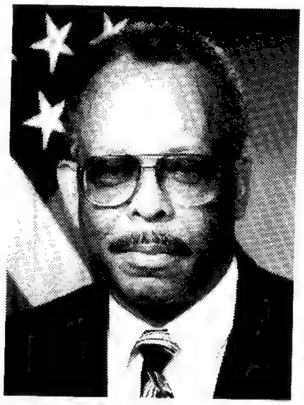
General Short was promoted to Major General when he became Information Systems Command Deputy Commander on 7 September 1988. General Short then assumed command of the Information Systems Command in June 1990 and was promoted to Lieutenant General at the same time.

In August 1991, General Short became Director, Defense Information Systems Agency/Manager, National Communications System.

His awards and decorations include: the Distinguished Service Medal; Legion of Merit; Bronze Star Medal with oak leaf cluster; Meritorious Service Medal with oak leaf cluster; National Defense Service Medal; Vietnam Honor Medal; and the Parachutist and Air assault Badges.

General Short and his wife, the former Rosalin Reid, of Orange, New Jersey, have one son, Stanley, and on daughter, Daniele.

THE HONORABLE EMMETT PAIGE, JR.



Assistant Secretary Of Defense For Command, Control, Communications and Intelligence

Mr. Emmett Paige, Jr., is the Assistant Secretary of Defense for Command, Control, Communications and Intelligence (ASD(C3I)).

He retired from the U.S. Army as a Lieutenant General on August 1, 1988, after almost 41 years of active service in Communications-Electronics business.

He enlisted in the U.S. Army in August 1947 as a Private at the age of 16, dropping out of high school to do so. After completing Signal Corps Officers Candidate School in July 1952 he was commissioned as a 2nd Lieutenant.

He commanded the 361st Signal Battalion in Vietnam and later the 11th Signal Group at Fort Huachuca, Arizona. Additionally, he served two tours with the Defense Communications Agency.

In 1976, he was promoted to Brigadier General and assumed the U.S. Army Communications-Electronics Engineering and Installation Agency at Fort Huachuca, Arizona, and concurrently the U.S. Army Communications Systems Agency at Fort Monmouth, New Jersey.

In 1979, he was promoted to Major General and assumed command of the U.S. Army Communications Research and Development Command at Fort Monmouth, New Jersey. Then in 1981, he assumed command of the U.S. Army Electronics Research and Development Command, located at the Harry Diamond Laboratories, Adelphi, Maryland.

In 1984, he was promoted to Lieutenant General and assumed command of the U.S. Army Information Systems Command where he served until his retirement.

Following his retirement from the U.S. Army, until his appointment as the ASD(C3I), Mr. Paige was the President and Chief Operating Officer at OAO Corporation, an Aerospace and Information Systems Company in Greenbelt, Maryland.

Mr Paige is a graduate of the U.S. Army War College. He has been honored as a "Distinguished Alumnus" of both the University of Maryland, University College, where he obtained his Bachelors degree; and Penn State where he received his Masters degree and in 1993; was selected as an Alumni Fellow. He was awarded an Honorary Doctor of Law degree from Tougaloo College, Tougaloo Mississippi. He has also been awarded an honorary Doctorate from the University of Maryland, Baltimore County.

He has been awarded the Army Commendation Medal, the Meritorious Service Medal, the Bronze Star for Meritorious Service in Vietnam, the Legion of Merit with two oak leaf clusters, and the Distinguished Service Medal with one oak leaf cluster.

Mr. Paige was selected as the Chief Information Officer of the Year in 1987 by Information Week Magazine, In 1988, he was selected for the coveted Distinguished Information Sciences Award winner by the Data Processing Management Association for outstanding service and contributions internationally to advancements in the field of Information Sciences.

He is married to the former Gloria McClary and has three children, Michael, Sandra, and Anthony.

RALPH E. CRAFTS

President
Ada Software Alliance

Ralph Crafts serves as the President of the Ada Software Alliance(ASA), the Ada industry's trade association whose mission is to promote the use of Ada in all software markets. Crafts has more than 29 years of executive and management experience in technology-based organizations, in both the military and commercial sectors. Since 1981, he has focused exclusively on the Ada marketplace, having been directly involved in the competition, procurement, and management of more than $2 billion worth of Ada contracts. In 1986, Crafts formed his own company, Software Strategies and Tactics, Inc. (SS&T), to provide Ada-related consulting services to the international software community. He is the editor and publisher of **Ada Strategies**, the Ada industry's only commercial international newsletter that focuses on Ada management issues, trends, case histories, and usage in both government and commercial applications.

Crafts was the Chairman of the first international TRI-Ada Conference and Exposition (in 1988), and served as the Public Information Coordinator of the national SIGAda (Special Interest Group on Ada) from 1988-89. He is an active member of the SIGAda and IEEE Computer society, and a member of the IEEE'S COPP (Committee on Public Policy).

Crafts' background includes eight years as a tactical jet pilot in the U.S. Marine Corp, and several years in a Fortune 500 MIS environment.

COMPARISON OF DESIGN DIVERSITY TECHNIQUES FOR FAULT-TOLERANT SOFTWARE DEVELOPMENT

Don M. Coleman
Ronald J. Leach
Department of Systems & Computer Science
Howard University
Washington, D.C. 20059

ABSTRACT

This paper describes the results of an experiment designed to evaluate the effectiveness of a set of techniques for developing fault tolerant software. The techniques are applied to three different types of programming applications: one with symbol manipulations, one with integer operations, and a third focused on floating point operations. Execution overhead is used to compare the relative performance of each technique with a given type of application program. Results are presented which show the relative effectiveness of each technique along with indications regarding the effectiveness of each technique in a particular application domain.

1. INTRODUCTION

Several techniques for development of fault-tolerant software have been proposed [3, 14]. Experiments have been performed evaluating and verifying the effectiveness of some of these techniques [9]. However, even when such techniques can be implemented, for a programming application, they may not be practical. In particular it is of interest to determine the "cost" of implementing such techniques. This cost, which can be defined as execution overhead, is important for certain critical applications. Such applications may have execution time as a critical requirement, examples include real-time systems with hard deadlines. In this paper we report on an experiment in which three techniques based on different uses of the design diversity approach to software fault tolerance are evaluated.

Design diversity [11, 13] is the software analog of the hardware notion of redundancy. The idea is to have multiple independent developments of software systems based on the same specification. It is assumed that the independent development efforts will minimize the likelihood of correlated design errors; that is, independently developed versions having similar errors at similar check points. The three methods considered here are: N-Version Programming (NVP) [3, 11], Recovery Block (RB) [14], and Consensus Recovery Block (CRB) [9].

The NVP method requires that N (2 or more) independent software modules, called versions, are generated from the same initial specification. Independent generation means that the programming efforts are carried out by individuals or groups of individuals who do not interact with respect to the programming process. Whenever possible, different algorithms and programming languages or translators are used in each effort. The intention is to have as much independence as possible between the N different software implementations each of which constitutes a solution to the problem. The NVP method requires the polling of the N independent processes at intermediate points (break points) of the algorithm. At each break point, a voter process determines the "correct" state for the computation. The method for determination of the "correct" state is by a majority vote among the N versions. Hence, correctness is really a consistency check; i.e., the most frequently occurring value at a break point becomes the "correct" state. In addition, it is assumed that differences between the processes are not the result of errors in the voter; i.e., the voter should either be fault tolerant or be subject to a formal proof of correctness. The computation proceeds until either all break point have been checked and the computation ends, or a voting inconsistency occurs and the program is aborted with an appropriate error message. The number of voting process is an open question and application specific; but for real-time applications it appears that the number will be small [6].

The RB method also requires the development of multiple independent versions similar to the NVP scheme; in addition, an acceptance test is required. RB is implemented with a primary module and several alternates, where the primary version is assumed to be the most reliable, and the alternates are chosen in order of decreasing reliability. The method requires that the results from the primary module (version) be checked at each break point against the acceptance test. The state of the computation is saved at each break point. The alternate versions are not executed until the primary module fails an acceptance test. (The acceptance test is generally a quick check; passing of an acceptance test does not guarantee correctness of the program.) At this point, the first alternate is invoked and becomes the primary module; it is initialized at the most recent break point where the state was known to be correct (i.e., the acceptance test was passed). If at some point the first alternate fails an acceptance test, the process is repeated, with the second alternate brought in as the primary module. The computation proceeds in this manner until all break point have been checked and the computation ends, or all alternates have failed the acceptance test which means the computation is aborted with an appropriate error message. The RB method is a backward recovery technique since at the point where the acceptance test is failed, the computation is rolled back to the last point at which the computation was known to be correct.

The CRB method is a hybrid of NVP and RB [9]. It can be considered a two phased approach with the first phase representing a NVP implementation and the second phase is an implementation of RB. When there is an inconsistency found at a break point in the first phase (NVP), the computation is then transferred to phase two (RB) with the appropriate initialization. At this point either the RB module(s) pass all acceptance tests and the computation end normally or the computation is aborted with an appropriate error message.

2. PROBLEMS USED FOR THE EXPERIMENT

In order to determine the effectiveness of each of the three fault-tolerance methods (NVP, RB, CRB), experiments were performed to measure the overhead associated with each technique. Overhead is defined here as the additional execution time required by the fault-tolerant implementation. Three different types of programming problems were implemented in Verdix Ada using each method. The objective was to determine if a particular method was more effective for a given type of problem. Each method required development of a procedure for measuring and collecting timing data.

The first programming problem, CALCULATOR, was one which required a considerable amount of symbol manipulation; it was a program designed to emulate a desk calculator which could perform arbitrary precision arithmetic. The desk calculator used a stack for the storage of intermediate results. The contents of the stack were pointers to structures pointing to arrays of characters that represented the values of the operands of the desk calculator. The operators and operands for this problem were given in data files with one operator or operand per line. Thus the size of the data file (as measured by the number of lines) indicated the number of activities of the stack since each operand is pushed onto the stack and each operator required stack access. The sequence of values associated with the stack operation were monitored to determine the "correctness" of a given program.

The second programming problem, SORT, required the sorting of arbitrary lists of randomly generated integers. Bubble sort, insertion sort, integer sort, and quicksort programs were developed for this problem.

The third programming problem, FFT, required a considerable amount of floating point arithmetic. A set of programs was developed to compute the Fast Fourier Transform [16] of N given complex points.

3. NVP IMPLEMENTATIONS

For the NVP implementations we selected a set of break points; at each break point the values (from the independent versions) were passed to the voter for comparison of the results.

For CALCULATOR, a natural break point was determined by each action involving an item on the top of the stack, that is, each time a version handles an item at the top of the stack, that item was compared to determine if each version was handling the same item at the same point in the algorithm. Voting was accomplished by an exact matching of the items.

For SORT, we did not have a efficient method for selecting breakpoints. It was not useful to compare partially sorted strings since as new elements were added, the partially sorted sequences could change. We implemented a voting scheme for the independent sorting programs; comparisons were made after the total completion of a sorting sequence. For sorting sequences a exact matching of the integers in the sequence is the criteria used in voting.

For FFT, a natural break point was after the computation of each component of the transform; since each component of the transform is represented by a complex number (two real numbers representing the real and imaginary parts) the voter will have to deal with comparisons of reals or floating point numbers. This presents another kind of problem for NVP since exact comparisons are not feasible for floating point numbers. Therefore it was necessary to establish, a priori, a range of acceptable differences between the values provided to the voter by the various independent versions at corresponding break points; i.e., at the i^{th} break point we have:

d_i = the maximum acceptable difference between the value provided by version X at the break point,

e_i = the acceptable value at the break point,

x_i = the value of the computation of version X at the break point,

Hence a value is acceptable to the voter when $|x_i - e_i| < d_i$.

To implement the FFT using the NVP fault-tolerant technique we developed a priori a table of values (e_i' s) for use in the voting scheme. The values were computed independently and each e_i was a pair of real (floating point) numbers representing the real and imaginary part of the i^{th} component of the transform.

4. RECOVERY BLOCK IMPLEMENTATIONS

The RB method requires development of an acceptance test for testing the output of the primary routine at each check point. The check points selected for each of the three programming problem were the same as the "break points" selected in the NVP implementations.

In CALCULATOR, the acceptance test for RB was implemented by providing an independently computed "acceptance table" with a sequence of values to be used with the primary version. The test again was an exact matching of characters between the value from the primary version and the value in the table.

When SORT was implemented using the RB technique, two acceptance tests were used. One acceptance test (checksum) simply compared the sums, before sorting began, to the sums after sorting had completed. The other acceptance test (casting out nines) used modulo 9 arithmetic, then compared the results of both modulo sums before sorting began to the sum after sorting completed. These acceptance tests were not concerned with the actual ordering of the elements (because of performance considerations), they simply checked the sum at a specified break point and compared it with the original sum which was computed before the sorting began.

When FFT was implemented with the RB technique used an acceptance test similar to the scheme used in voting with NVP. That is, at each check point, the output of the primary algorithm was compared with a value from a table of acceptable values. When that value was within the acceptable range, the computation continued using the primary algorithm; otherwise the secondary routine was entered, the computation was "rolled back," i.e., the last correct state of the primary algorithm was restored and the computation proceeded from that point using the secondary algorithm. When the secondary algorithm failed the acceptance test, the computation was terminated.

5. CONSENSUS RECOVERY BLOCK IMPLEMENTATIONS

The CRB is a cascaded scheme with the front end provided by an NVP implementation and the back-end provided by a recovery block. For this experiment only two versions were used for the NVP front end. The back ends (RB) for each problem consisted of a primary and a single secondary module. Break points and acceptance tests were similar to those used in the previous implementations.

6. RESULTS

For each problem and each fault-tolerant method, test runs were completed for five data sets. The numbers presented in Table 1 are the average relative execution times for ten runs with a given data set.

For example, the total N-version execution time for two versions and data file number 1, relative to the time to execute one version is 1.20. This means that 20% more time was required to execute the algorithm with the two-version calculator implementation. Thus there was an additional overhead of 20% required to implement fault tolerance for this application.

We also studied the relationship between the overhead required for each fault-tolerant technique and the "load." For CALCULATOR, the load is defined by the number of calculator activities, which, in turn, is defined by the number of activities of the stack since each operand is pushed onto the stack and each operator requires stack access. Thus the size of the data file (as measured by the number of lines) determines the size of the "load." The "load" would be the array size for both FFT and SORT.

For this experiment the data files are numbered from 1 to 5 in increasing size or increasing load. Note that for each N (2, 3, 4, 5, and 6) in the NVP experiment, the overhead is relatively stable with load as shown by the relatively horizontal lines given in Figure 1 where we show overhead vs load.

An examination of Figures 2 and 3 show that for the other programming examples we did not have as stable behavior as we increased load . In Figure 2 overhead varies from 33% to 200% for N=2 at a load of 2 and 4 respectively. Figure 2 also shows that for the larger arrays of data points, there was less variation of overhead with load. For the SORT implementations, as shown in Figure 3, there is almost a linear increase of overhead as the load increases. The average overheads are higher for both SORT and FFT than for CALCULATOR for a given number of versions.

It appears that the NVP is more efficient (has lower overhead) when the voter is handling data which require an exact match (19 %) as opposed to integer (63%) or real number (114%) arithmetic.

In the Table, the total RB execution time is relative to the time to execute one version is shown. For example, for data file 1 the relative execution time for CALCULATOR with "No Faults" is 1.09, which implies an overhead of 9% associated with the RB fault-tolerance implementation for this data set. The column labeled "With Faults" indicates the average results when the algorithm's acceptance test is failed and the secondary block is entered. When this happens, the computation is restored to its last correct state and the process is continued with the secondary module and the same acceptance test. For this experiment the "best" module was used as the primary module hence the recovery module usually will not meet the acceptance test as well, therefore, the computation is terminated. For data file 1, the overhead was 90%.

Figures 5, 6, and 7 show the average overhead as a function of load (as defined above). Each graph has curves labeled "Recovery" and "No Recovery." The label "No Recovery" means that the computation proceeded with the primary module passing each acceptance test; while the curve labeled "Recovery" represents the results when the primary module did not pass an acceptance test and the recovery block was entered. Figure 5 shows that the average overhead for CALCULATOR is stable as a function of load both for the "Recovery" and "No Recovery" modules. For each of the programming examples we find the "No Recovery" results to be fairly stable as a function of load. There is an irregularity with FFT where there is an relatively high overhead due to the extremely small time required to find the transforms for a two component array. Note that when recovery is required, the overhead is large compared to the NVP methodology. With RB, we have a backwards recovery technique which requires restoring the computation to its last correct state; this appears to require more overhead time than a forward recovery process such as NVP. The SORT implementations, as the "No Recovery" curve shows, required very little overhead. This can be attributed to the simplicity of the acceptance tests for the SORT implementation and the fact that only one routine is executed.

Of course, there is a trade-off between simplicity of the test and the effectiveness of a test in discovering an error. When the SORT recovery block was entered we find that we have the overhead growing linearly with load. The time to restore the computation to the last correct state depends of the length of the array of numbers being sorted.

Recall that the CRB method is a two-phased approach to fault-tolerance. In our case, the first phase is a NVP implementation and the second phase is the RB implementation. There is a much larger overhead than for either NVP or RB. Figures 8, 9, and 10 show that the the three programming problems gave somewhat stable behavior as a function of load. However, for the worst case of SORT, the overhead of CRB is almost 23 times higher than RB. The preliminary conclusion is that CRB would not be an effective approach because of these unacceptable levels of overhead.

7. CONCLUSIONS

The purpose of this experiment was to evaluate the relative effectiveness and efficiency of the three fault-tolerant techniques N-Version Programming, Recovery Block, and Consensus Recovery Block. The measure of efficiency used was the overhead or additional computational time required when fault-tolerant code was added to the program. The overall observation was that the most efficient technique was RB with the least efficient being CRB, at least when combined with NVP.

From Figure 11 we observe that CALCULATOR is the best suited application for these three fault-tolerant techniques. Figure 12 is a direct comparison of NVP and RB in terms of average percent overhead over all loads. The figure shows that for this set of problems the recovery block method gave the best results. This implies that the acceptance test for this set of problems was more efficient than the voting. This follows because, with NVP, at least two algorithms will have to be executed for comparison at each break-point, while with RB only one algorithm is executed at a break-point and a second algorithm is executed only if the acceptance test fails. Hence, in the case where the comparison is made between programs where no errors are found the RB method should require less overhead. When errors occur, then the overhead of RB becomes excessive relative to NVP because of the roll-back required for transfer to the secondary routine associated with the recovery. This experiment suggests that if a cascaded technique such as CRB is used, the first phase should be RB.

Below we summarize some specific lessons learned as a result of this work:

- In general, RB is the least costly method

- A problem using the NVP technique should be decomposable into a sequence of break points at which it is feasible to make comparisons about intermediate accuracy .

- Problems dealing with symbols, where voting and acceptance tests are based on exact comparisons, are best suited for application of these fault-tolerant techniques.

- Programming problems with real numbers must have a way of establishing a range of acceptable results.

- The application of any of these methods requires one of two assumptions: the development of diverse algorithms is done in independent environments and that the voting and acceptance codes are assumed to be mature and error free.

- A backwards recovery technique such as RB which requires restoring the computation to its last correct state appears to require more overhead time than a forward recovery process such as NVP.

- There is a trade-off between simplicity of the test and the effectiveness of a test in discovering an error.

- The preliminary conclusion is that CRB would not be an effective approach because of these unacceptable level of overhead.

- For problems which are unlikely to have faults, the RB method will require less overhead.

- For problems with hard timing constraints, if NVP is the method used, N=2 is the likely number of versions.

- If a CRB technique is to be implemented and performance is essential, the first phase should be RB.

8. FUTURE WORK

This experiment has shown the RB method to offer the most promise (least cost in overhead) for problems of the type reviewed here. The next experimental step involves using the RB method with a real-time system with hard deadlines. The major task is the development of acceptance tests which would allow deadlines to be maintained. We will also examine how a FT implementation affects the probability models used to predict errors in software. It is not clear whether we change parameters in a reliability model or change the model structure itself.

ACKNOWLEDGEMENT: This work was partially supported by Wright Laboratory under Contract Number F33615-91-C-1758.

BIBLIOGRAPHY

[1] Abbott, R. J., "Resourceful Systems for Fault Tolerance, Reliability, and Safety," ACM Computing Surveys, Vol. 22, No. 1, March 1990.

[2] Anderson, and P. A. Lee, Fault Tolerance Principles & Practice, Prentice-Hall International, Inc., 1981.

[3] Avizienis, A. and J. P. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," IEEE Computer, pp. 67-80, August 1984.

[4] Belli, F., and P. Jedrzejuwicz, "Fault-Tolerance Programs and Their Reliability," IEEE Transactions on Reliability, Vol. 39, No.2, pp. 184-192, June 1990.

[5] "Chapter 5: Software Fault Tolerance," 1987 IEEE, pp. 247-256.

[6] D. M. Coleman, and R. J. Leach, "Performance Issues in C Language Fault-Tolerant Software," Computer Languages, Vol. 14, No. 1, pp.1-9, 1989.

[7] Gault, J. W. , McAllister, D. F., and K. Scott, Fault Tolerant Software Reliability Modeling," IEEE Transactions on Software Engineering, Vol. SE-13, No. 5, pp. 582-592, May 1987.

[8] Goel, A. L., "Software Reliability Models: Assumptions, Limitations, and Applicability, IEEE Transactions on Software Engineering, Vol. SE-11, No. 12, pp.257-269, December 1985.

[9] Goel, A. L., and N. Mansour, "Software Engineering for Fault-Tolerant Systems," Final Technical Report, Rome Laboratory, March 1991.

[10] Jalote P., and J. M. Purtilo, "An Environment for Developing Fault-Tolerant Software," IEEE Transactions on Software Engineering, Vol. 17, No. 2, pp. 153-182, Feb. 1991.

[11] Kelly, J. P. J. , McVittie, T., I., and W. I. Yamamo, "Implementing Design Diversity to Achieve Fault Tolerance," IEEE Software, pp. 61-71, July 1991.

[12] Lee, A., "A Reconsideration of the Recovery Block Scheme," The Computer Journal, Vol. 21, No. 4, pp. 306-310, January 1978.

[13] Murugesan, S., and Singh, A., "Fault-Tolerant Systems," IEEE Computer, pp. 15-17, 1990

[14] Randell, B., "System Structure for Software Fault Tolerance," IEEE Transaction on Software Engineering, Vol. SE-11, No. 2, June 1975.

[15] Horning, J. J. A program structure for error detection and recovery. Lecture Notes in Computer Science (Vol. 16). Springer-Verlag, 1974, pp. 171-187.

[16] Press W. H. et al., Numerical Recipes: The Art of Scientific Computing, University of Cambridge Press Syndicate, 1987, pp. 381-453.

	NVP		Recovery Block		Concensus	Recovery Block
DATA FILE	VERSIONS		No Faults	With Faults	No Faults	With Faults
	N=2	N=3				
1	1.20	1.44	1.09	1.9	3.89	8.44
2	1.23	1.47	1.18	1.8	2.86	4.7
3	1.22	1.65	1.14	1.69	1.77	3.02
4	1.19	1.31	1.17	1.79	1.38	2.3
5	1.10	1.28	1.13	1.72	1.59	1.76
Average Overhead	**1.19**	**1.43**	**1.14**	**1.78**	**2.3**	**4.05**

CALCULATOR SIMULATION

	NVP		Recovery Block		Concensus	Recovery Block
	VERSIONS		No Faults	With Faults	No Faults	Faults
Array Size	N=2	N=3				
2	3	4.5	4.5	10	5	18
4	1.33	5.33	2	4.33	2.17	6.67
32	2.44	4.38	1.63	3.38	3.44	7.81
64	2.08	3.44	1.25	2.53	3.14	6.56
128	2.11	3.38	1.19	2.33	3.22	6.42
256	1.88	2.81	1.01	1.94	2.9	5.55
Average Overhead	**2.14**	**3.97**	**1.93**	**4.08**	**3.31**	**8.5**

FFT CALCULATIONS

	NVP					
	VERSIONS		Recovery Block		Concensus	Recovery Block
No. Sorted	N=2	N=3	No Faults	With Faults	No Faults	With Faults
10	1.14	1.28	1.4	3.6	25.8	27
30	1.43	1.91	1.2	5	22.4	40.5
50	1.6	2.24	1.36	10.55	31.18	51.18
70	1.73	2.7	1.19	13.38	29.81	57.69
100	2.23	2.96	1	17.78	30	49.91
Average Overhead	**1.63**	**2.22**	**1.23**	**10.06**	**27.84**	**45.26**

SORTING RESULTS

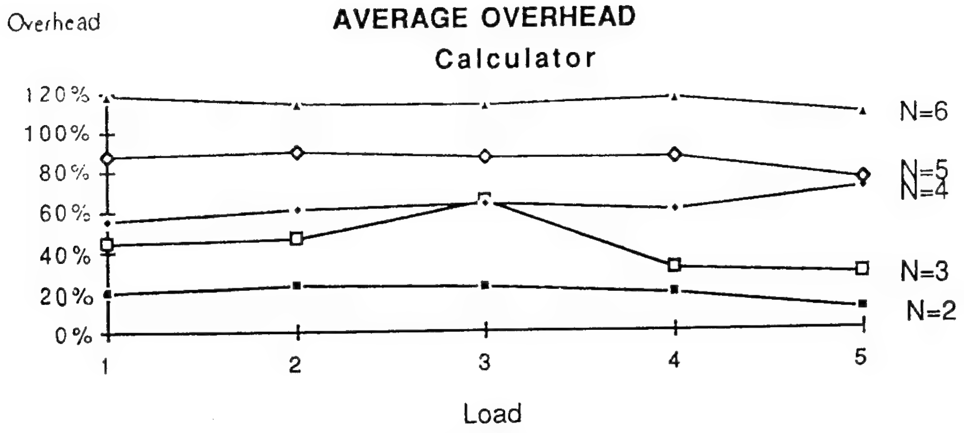


FIGURE 1

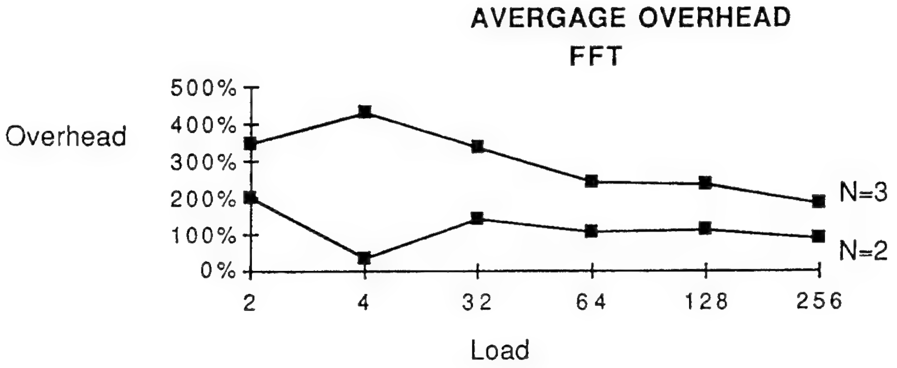


FIGURE 2

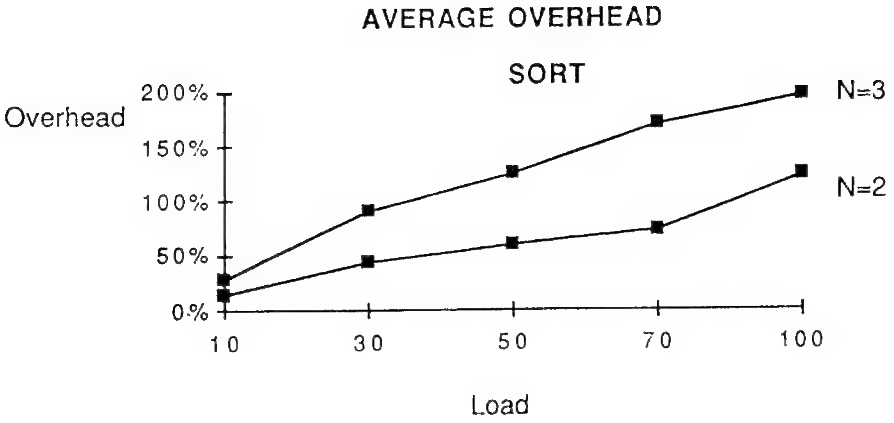


FIGURE 3

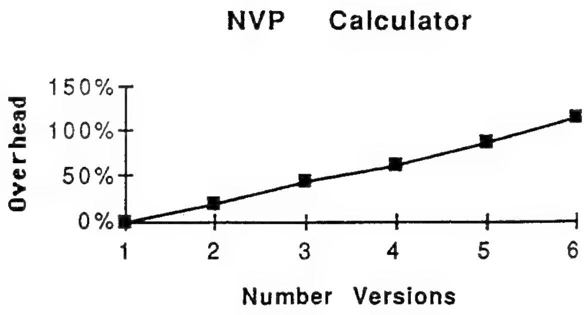


FIGURE 4

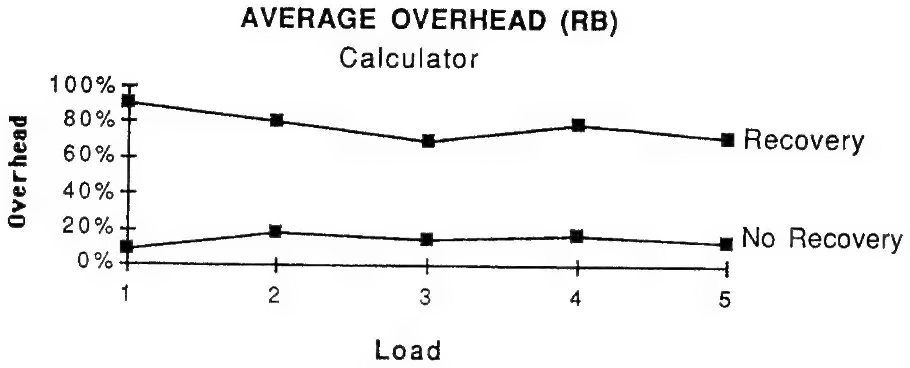


FIGURE 5

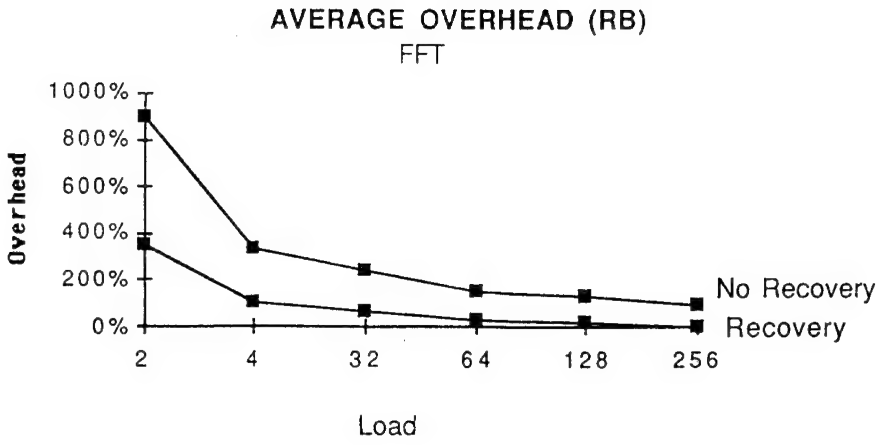


FIGURE 6

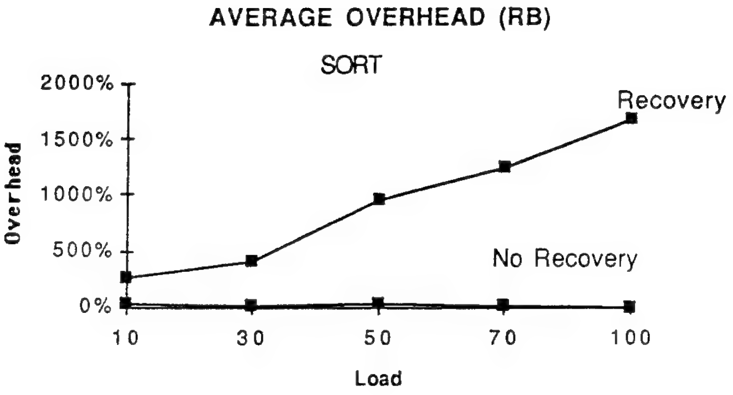


FIGURE 7

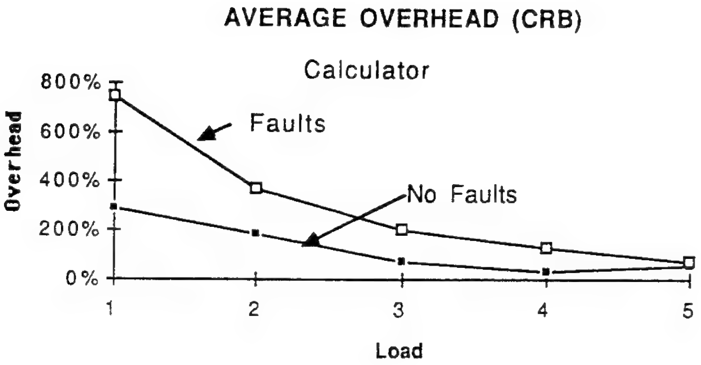


FIGURE 8

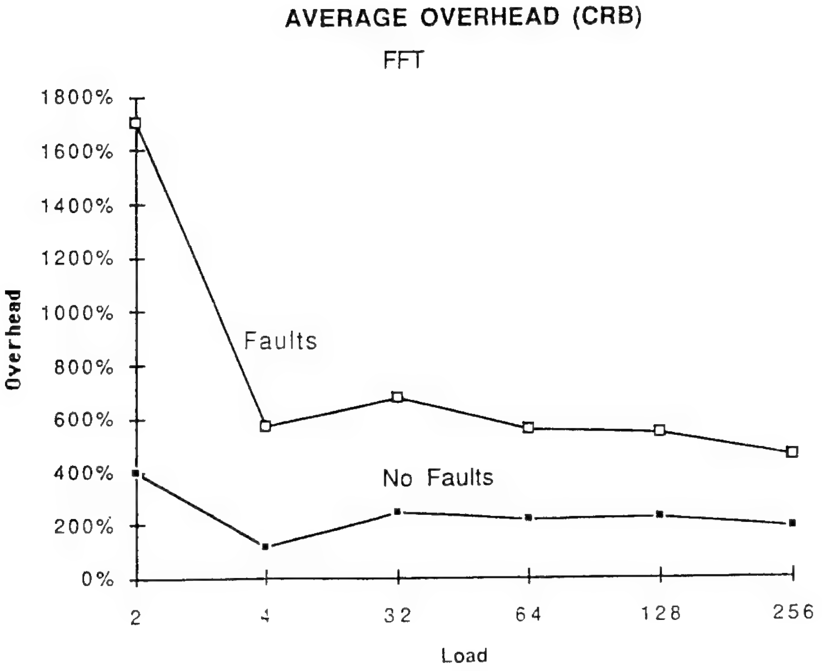


FIGURE 9

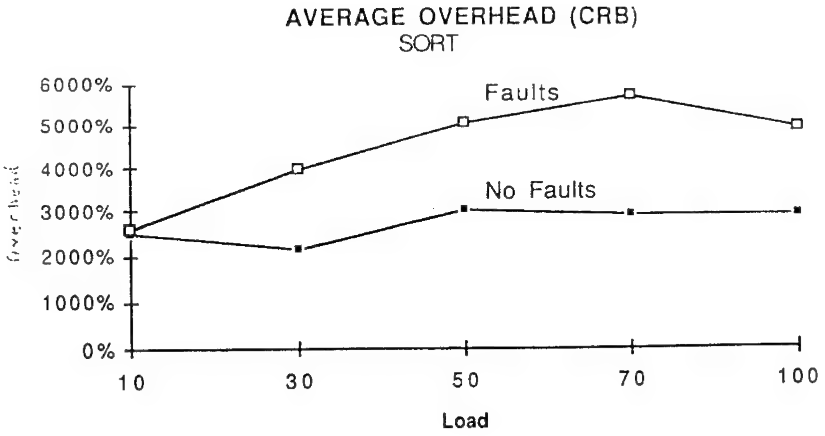


FIGURE 10

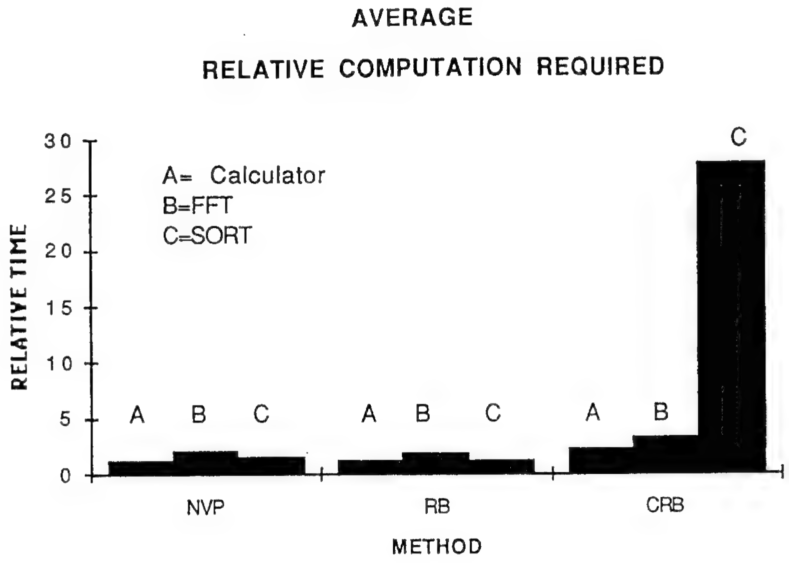


FIGURE 11

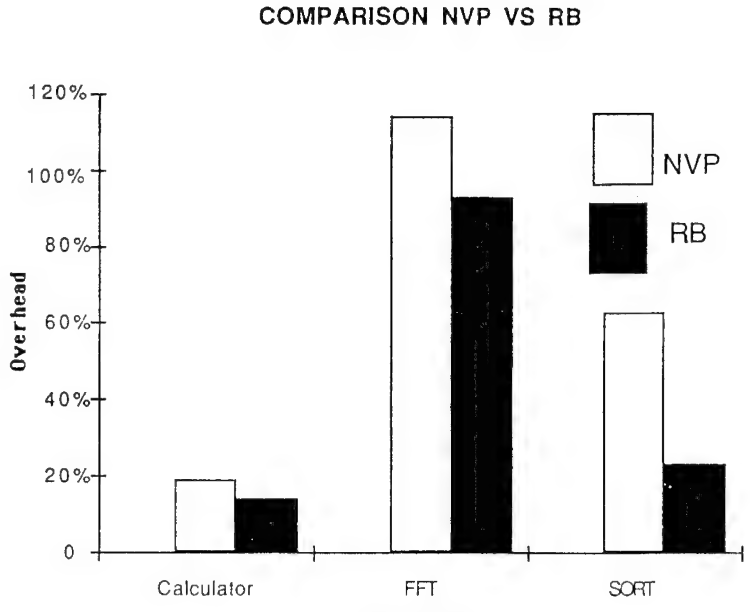


FIGURE 12

An Overview of the Ada Semantic Interface Specification (ASIS)

Bill Fay
u7c31@gsde.hso.link.com

Viktor Ohnjec
vo@rational.com

ABSTRACT

The Ada Semantic Interface Specification (ASIS) defines an interface that provides access to the syntactic and semantic information created by an Ada compiler. This interface greatly simplifies the creation of CASE tools that may be used for metrics gathering, reverse engineering, style checking, testing and other applications.

This paper will give a high level overview of ASIS, including a brief history, and will illustrate key structures and operations defined by the standard.

HISTORY

ASIS was conceived on the Software Technology for Adaptable and Reliable Systems (STARS) project when a handful of Telesoft engineers responded to the needs of the customer and provided an interface to the semantic information created and stored by the Ada compiler. Meanwhile, Rational was providing what they referred to as the Ada LRM (Language Reference Manual) interface on their R1000 line of Ada software development environments. The ASIS standard was born when Rational donated much of the LRM interface specification to the public domain. It is not surprising, then, that Telesoft and Rational were the first Ada compiler vendors to provide the ASIS interface as a commercial product.

OVERVIEW

The ASIS interface provides semantic information on three main levels or abstractions. As described in section 10.4 of the LRM, the *program library* (level 1) contains the source code and any related compiler-generated information associated with *compilation units* (level 2). In the third level, the compilation units are composed of individual pieces or *elements.*

Program Library

Operations provided for program libraries include 'Associate' which binds a particular library to the program using the ASIS interface and 'Open' and 'Close' which perform the obvious. The program library interface also provides functions that return information such as the library's name and whether the library is open or has associations. These operations are provided in a package named 'Asis_Libraries'.

Compilation Unit

Package 'Asis_Compilation_Units' encapsulates the operations on a compilation unit. The kinds of compilation units include package declaration, subprogram declaration, generic declaration, package body, subprogram body and generic instantiation. Functions that return attributes of a compilation unit include 'Kind', 'Time_Of_Last_Update', 'Subunits', 'Dependents', 'Supporters' and 'Is_Obsolete'.

Elements

Elements are either single syntactic items (also known as 'tokens') or collections of the former. For example,

```
      Pi : constant := 3.1415926;
```

is a constant declaration element composed of a name and a real literal. Other types of elements include type declarations, expressions, statements, and 'with' and 'use' clauses.

ASIS PACKAGES

The ASIS standard is implemented with seventeen portable and two non-portable packages. Several of the portable packages are described below.

Package Asis

This is the base package for the ASIS interface and as such, it provides visibility to the remaining portable types and operations. It is, for the most part, a series of subtype declarations and rename clauses.

Package Asis_Compilation Units

This package provides various operations on compilation units, such as:

```
function Time_Of_Last_Update
  (Compilation_Unit : in Asis.Compilation_Unit)
    return Asis.Asis_Time;

function Enclosing_Compilation_Unit
  (Element : in Asis.Element)
    return Asis.Compilation_Unit;
  -- The compilation unit that contains
  -- 'Element'.

function Unit_Declaration
  (Compilation_Unit : in Asis.Compilation_Unit)
    return Asis.Declaration;
  -- The declaration of the compilation unit.
  -- This is the 'gateway' to the declaration
  -- package.

function Context_Clause_Elements
  (Compilation_Unit : in Asis.Compilation_Unit;
   Include_Pragmas  : in Boolean := False)
    return Asis.Context_Clause_List;
  -- The list of 'with' and 'use' clauses.

function Is_Subunit
  (Compilation_Unit : in Asis.Compilation_Unit)
    return Boolean;
  -- Returns True if the compilation unit is
  -- a subunit.

function Subunits
  (Parent_Unit : in Asis.Compilation_Unit)
```

```
      return Asis.Compilation_Unit_List;
   -- The subunits defined for the compilation
   -- unit.

function Dependents
   (Compilation_Unit  : in Asis.Compilation_Unit;
    Compilation_Units : in
                  Asis.Compilation_Unit_List;
    Library : in Asis.Library)
      return Asis.Compilation_Unit_List;
   -- List of compilation units dependent
   -- upon the compilation unit.

function Is_Obsolete
   (Compilation_Unit : in Asis.Compilation_Unit)
      return Boolean;
   -- Returns true if the internal information
   -- for the compilation unit has been obsoleted
   -- by a compilation of a unit above it in its
   -- closure.
```

Package Asis_Elements

Function 'Kind' is used to determine the kind of the element as one of:

```
type Element_Kinds is
      (A_Pragma,
       An_Argument_Association,
       A_Declaration,
       An_Entity_Name_Definition,
       A_Type_Definition,
       A_Subtype_Indication,
       A_Constraint,
       A_Discrete_Range,
       A_Discriminant_Association,
       A_Variant_Part,
       A_Null_Component,
       A_Variant,
       A_Choice,
       A_Component_Association,
       An_Expression,
       A_Statement,
       An_If_Statement_Arm,
       A_Case_Statement_Alternative,
       A_Parameter_Association,
       A_Use_Clause,
       A_Select_Statement_Arm,
       A_Select_Alternative,
       A_With_Clause,
       An_Exception_Handler,
       A_Representation_Clause,
       A_Component_Clause,
       Not_An_Element );

function Element_Kind
    (Element : in Asis.Element)
      return Asis_Elements.Element_Kinds;
```

The generic procedure 'Traverse_Element' is extremely useful for visiting elements in compilation units. 'Pre_Operation' and 'Post_Operation' are procedure provided by the user and are called as each element is visited, upon entry and exit, respectively. A data object declared as 'State_Information' is passed into 'Traverse_Element' which is in turn passed to each call of 'Pre_Operation' and 'Post_Operation'.

```
type Traverse_Control is
       (
       Continue,
       Abandon_Children,
       Abandon_Siblings,
       Terminate_Immediately
       );  -- used by 'Traverse_Element'

generic

  type State_Information is
       limited private;

  with procedure Pre_Operation
     (Element : in Asis.Element;
      Control : in out
                Asis_Elements.Traverse_Control;
        State : in out State_Information) is <>;

  with procedure Post_Operation
     (Element : in Asis.Element;
      Control : in out
                Asis_Elements.Traverse_Control;
        State : in out State_Information) is <>;

  procedure Traverse_Element
     (Element : in Asis.Element;
      Control : in out
                Asis_Elements.Traverse_Control;
        State : in out State_Information);
```

Package Asis_Declarations

Function 'Kind' is used to determine the kind of the declaration as one of:

```
type Declaration_Kinds is
       A_Variable_Declaration,
       A_Component_Declaration,
       A_Constant_Declaration,
       A_Deferred_Constant_Declaration,
       A_Generic_Formal_Object_Declaration,
       A_Discriminant_Specification,
       A_Parameter_Specification,
       An_Integer_Number_Declaration,
       A_Real_Number_Declaration,
       An_Exception_Declaration,
       An_Enumeration_Literal_Specification,
       A_Loop_Parameter_Specification,
       A_Full_Type_Declaration,
       An_Incomplete_Type_Declaration,
       A_Private_Type_Declaration,
       A_Subtype_Declaration,
       A_Package_Declaration,
       A_Package_Body_Declaration,
       A_Procedure_Declaration,
       A_Procedure_Body_Declaration,
       A_Function_Declaration,
       A_Function_Body_Declaration,
       An_Object_Rename_Declaration,
       An_Exception_Rename_Declaration,
       A_Package_Rename_Declaration,
       A_Procedure_Rename_Declaration,
       A_Function_Rename_Declaration,
       A_Generic_Package_Declaration,
       A_Generic_Procedure_Declaration,
       A_Generic_Function_Declaration,
       A_Package_Instantiation,
       A_Procedure_Instantiation,
       A_Function_Instantiation,
       A_Task_Declaration,
       A_Task_Body_Declaration,
       A_Task_Type_Declaration,
       An_Entry_Declaration,
       A_Procedure_Body_Stub,
       A_Function_Body_Stub,
       A_Package_Body_Stub,
       A_Task_Body_Stub,
       A_Generic_Formal_Type_Declaration,
       A_Generic_Formal_Private_Type_Declaration
       A_Generic_Formal_Procedure_Declaration,
       A_Generic_Formal_Function_Declaration,
       Not_A_Declaration );

function Kind
  (Declaration : in Asis.Declaration)
    return
      Asis_Declarations.Declaration_Kinds;
```

Subprogram parameter declarations may be characterized via:

```
type Parameter_Mode_Kinds is
      (A_Default_In_Mode,
       An_In_Mode,
       An_Out_Mode,
       An_In_Out_Mode,
       Not_A_Parameter_Mode
       );

function Parameter_Mode_Kind
  (Declaration : in
         Asis.Parameter_Specification) return
     Asis_Declarations.Parameter_Mode_Kinds;
```

Package Asis_Statements

Function 'Kind' is used to determine the kind of the statement as one of:

```
type Statement_Kinds is
       (A_Null_Statement,
       An_Assignment_Statement,
       A_Procedure_Call_Statement,
       An_Exit_Statement,
       A_Return_Statement,
       A_Goto_Statement,
       An_Entry_Call_Statement,
       A_Delay_Statement,
       An_Abort_Statement,
       A_Raise_Statement,
       A_Code_Statement,
       An_If_Statement,
       A_Case_Statement,
       A_Loop_Statement,
       A_Block_Statement,
       An_Accept_Statement,
       A_Selective_Wait_Statement,
       A_Conditional_Entry_Call_Statement,
       A_Timed_Entry_Call_Statement,
       Not_A_Statement);

function Kind
  (Statement : in Asis.Statement)  return
     Asis_Statements.Statement_Kinds;
```

As an example of the level of granularity of the interface, an 'If' statement may be analyzed with the following:

```
type If_Statement_Arm_Kinds is
       (An_If_Arm,
       An_Elsif_Arm,
       An_Else_Arm,
       Not_An_If_Statement_Arm);

function If_Statement_Arm_Kind
 (Arm : in Asis.If_Statement_Arm) return
    Asis_Statements.If_Statement_Arm_Kinds;
```

References:

[1] Gary E. Barnes and the ASIS Working Group, "Ada Semantic Interface Specification," version 1.1.0, August 9, 1993.

[2] Rational, "Rational ASIS Concepts Guide," Revision 1.0, August 1993.

About the authors:

Bill Fay is a software engineer employed with CAE Link Space Operations in Houston, Texas and is currently working on the Space Station Verification and Training Facility (SSVTF) project at the Johnson Space Center. Mr. Fay leads the software engineering group which, among other responsibilities, is responsible for developing tools using the Rational R1000 LRM and ASIS interface. Mr. Fay holds a BS Electrical Engineering from Michigan State University.

Viktor Ohnjec is a software engineering consultant employed with Rational and is involved with several large-scale Ada projects including the Space Station Verification and Training Facility (SSVTF) and the Canadian Automated Air Traffic Control System (CAATS). Mr. Ohnjec received his BE Computer Engineering from Concordia University.

ADA'S ROLE IN PREVENTING DATA CORRUPTION : THEORY VS REALITY IN A LARGE AVIONICS SOFTWARE PROJECT

Mike Jenkins
Honeywell Air Transport Systems
Phoenix, Arizona

In large software projects, data corruption can be a serious threat to the reliability and performance of the system. Data corruption can be defined as "Data which abides by software checks until operated on, which then gives unexpected results". The most common form of data corruption is overflow of arrays and records. Previous languages have avoided data corruption by using data buffering and memory locking. In data buffering, data is written to multiple memory locations. Memory locking involves using the hardware or software to 'key' the memory. These methods may prevent data corruption, but both methods add computation time and require additional memory. Ada prevents corruption through the combination of data abstraction, data coupling, and proper use of parameter modes. Data is typed in small, relational groups, with separate types and data packages for each. Procedures are written which avoid using mode IN OUT. Input data is explicitly converted to a generic type within the parameter list. The result of the procedure is passed back using mode OUT, which decouples the input and output. This winning combination takes advantage of Ada's typing and abstraction strengths, portability, and reliability.

Avionics for today's large commercial aircraft are carrying more software than ever. That software is getting increasingly complex, and system boundaries are beginning to blur. Today's leading commercial avionics companies are heading toward an architecture which integrates most of the aircraft avionics into one processing center. Inputs from sources throughout the airplane are received by this center, which in turn sends them to various subsystems. Software for these architectures are now exceeding 500,000 lines of code, making them the largest avionics software systems ever.

The once familiar round dial analog instruments are being increasingly replaced with display units. Each unit, similar to a small television or PC screen, is capable of generating many formats. For example, a format might be a weather map, a flight director, or a pictorial presentation of the plane's fuel tanks and the fuel levels in those tanks. The job of the displays subsystem is to take these inputs and present them in an understandable, visual form to the flight crew.

The incoming data is totally diverse in its origin and function. To the displays, however, all data will be treated alike; all data is tested for validity before it is displayed. Organization and control of this diverse data is one of many reasons that companies choose Ada. Strong typing, error checking, and portability are others. The problem is to avoid data corruption without forcing the displays subsytem to treat all inputs uniquely. When this software is ready to be certified by the Federal Aviation Administration (FAA), one of the many tasks of the system will be to prove that its data handling and data transfer cannot be corrupted.

What is data corruption ? Why is it important ? Every software application must avoid 'bad data', or incorrect input. Fault tolerant software attempts to intercept unexpected data before it produces incorrect results. However, when data is corrupted, checks for bad inputs are useless. Data that has been corrupted looks like 'good' or valid data to the system, until the software operates on it. It means that data from one subsystem overwrites memory allocated for another subsystem's data. Typically, this occurs by writing into the wrong memory address, or overflowing the allocated space of a record or array. One way to do this is to write into the 11th location of a 10–element array. The 11th element, which does not exist, will be whatever the linker has assigned as the next memory address after the array.

A good working definition for corrupted data, then, would be "Data which abides by software checks until operated on, which then gives unexpected results." Data corruption within an avionics system has the potential to be disastrous, because of the large number of different systems sending the central processing system their inputs. The effects of this corruption can be minimal, such as displaying ground speed when true airspeed should have been displayed, or they can be lethal, such as sending an autopilot command a radio frequency.

Until Ada, most projects have tackled data corruption in two major ways – data buffering and data 'locking'. Data buffering involves writing the data received at one processing rate to a separate data name and memory location to be used by another rate. Most real time applications use a multiple rate architecture. There are some functions which must be computed or simulated at the fastest execution time the processor will allow. For instance, airplane trim changes very quickly, so a simulation of a trim system would require that it be computed at the maximum processing speed. However, processing all data at the fastest rate possible would require enormous throughput, which most real time applications do not possess. To accomplish both goals, applications process highly dynamic functions at the fast rate, and process all other functions at a slower rate. Data buffering is used when one rate needs the data produced by a different rate. This prevents problems associated with attempting to access data at one rate while it is being written into at another rate. The drawbacks are that it uses twice the data memory, adds critical execution time, and still does not protect the memory from accidental corruption within the same rate. Data buffering, then, is a technique used to ensure data is in the right place AT THE RIGHT TIME. A simpler definition for data corruption may be that data corruption occurs when data is not in the right place at the right time.

Data 'locking' involves using the processor to physically inhibit writes to selected sections of memory known as protected RAM (Random Access Memory). This usually means using a register or 'key' to allow data writes in the protected area. Legal access is obtained by first unlocking the memory. The write is performed, and the memory is again locked. Usually a local register is used as the key. Data must be transferred to the designated register, which is the only register allowed to write to the protected RAM. Again, this means an increase in computation time. It also means that the processor chosen must have the capability to lock certain areas of memory, or must be modified by the local hardware group to do so. At the very least, this leads to totally unportable code, since the processor hardware itself is included in the design and implementation.

With the introduction of Ada, data locking can be eliminated, and data buffering can be minimized. (Some buffering may still be required for multiple rate data access.) The first step in the solution rests in the simplest feature of Ada –

data typing. Ada has been hailed (and assailed) right from its start for its typing features. Because Ada is a rigidly typed language, it is possible to define the data in such a way that each variable, data set, or data structure receives its own type. This extreme has never been suggested as an actual method. However, typing data uniquely, although cumbersome and time consuming, requires no extra memory. Ada allocates memory space during data definition, so array and record overflow problems can be shown not to exist. Ada stringently checks for this kind of overflow during compilation, thus any code that is error free after compilation is guaranteed to be free of boundary problems. The exception to this rule lies in dynamic assignment of array indexes and unconstrained arrays. However, even these types must declare their bounds before assignments can be made to them. Run time assignment of array sizes are checked by Ada, and proper exception handling can prevent any unexpected results. Also, by typing each data or data set uniquely, Ada will guarantee that no data can be 'accidentally' written into another section of memory or data set. If the memory accessing scheme within the user code is faulty or fails, Ada will raise an error when data writes are attempted in incorrect locations. Thus, the built–in features of the language itself prohibit the problem of data corruption. No proof is required on the part of the user, provided the user types all the data uniquely. The drawbacks are 1)large effort to type all data uniquely; and 2)cannot use generics to process common functions.

The other extreme, of course, is to declare basic types for integer, real, boolean, and any other kind of data set to be used, and then to declare all data to be of these general types. This method literally strips Ada of most of its type checking ability. It is possible to reduce Ada to any other language, after all. So, somewhere in the middle lies a 'best' solution. For the displays subsystem, it was decided to type the data in related unit sets. All data in degrees Celsius were given a common_deg_c_type. This decision was based on the assumption that data with the same units would generally be related. All Celsius data would most likely be temperature data, for instance. Then, further grouping of data by function would help ensure that no chance for corruption occurred. Modules that handled cabin temperature would be separate from modules that handled exhaust gas temperatures.

Ada has still not quite solved the problem of data corruption. There is no free lunch. For displays, operating on the data means performing a common status check on all incoming data. Even though the data itself differs widely, ARINC (Aeronautical Radio, Inc) data buses used in most real time avionics applications assign a validity to each input. In the displays subsystem, that validity is incorporated into a record structure (Figure 1). The data comprises the first entry in the record, and the validity comprises the second. The data can be converted to a common type, or unchecked_conversion can be used to bypass the typing checks. To convert every piece of data to be manipulated adds enormous overhead to processing. To bypass the checks with unchecked_conversion invalidates the claim that Ada prevents data corruption. However, there is a healthy middle; one that allows the features of Ada to show the FAA that data corruption is avoided, and yet allows manipulation of common data.

The most straightforward method to employ is to package the data appropriately. First, a package specification that contains data types for logical or functional groups of data must be created. As mentioned above, all Celsius data would share a types package (Figure 2). Next, data packages (also package specifications) that assign data from the types packages must be generated (Figure 3). The reason for having separate types and data packages is twofold. The first is to distance the user from any knowledge of how the data is arranged. This is one form of data abstraction. The end user should only know the name of the data and its data package. Secondly, by having all the type definitions in their own packages, they can be WITHed and USEd separately. In places where equality checks are made, this allows Ada to perform overloading of "=", "<", ">", etc. rather than forcing the code to define those symbols for each new type. (Ada provides each new type with a set of operations for the data in that type. A function like "=" would have to be defined for each unique type.) Including package specifications for types and data using the WITH statement in Ada is done at program elaboration, which translates to initialization processing and does not add computation time to the program. Thus, we are free to group the types and data into as many functional areas as needed, without adding complexity from large numbers of packages. Also, by WITHing only those data packages that contain data needed by the subunits, data corruption between unlike data or subsystems is avoided. Code is seldom 100% bug free. Denying access to data not WITHed will help prevent the spread of errors.

The last problem to overcome is how to perform a simple operation or operations, such as a validity check, on large groups of unlike data. The displays subsystem tests a status field in every input that the databus provides. As each data system comes into the input module, its validity is to be verified by checking the assigned bus status. To do this, a procedure is created which has as its formal parameters an input variable of some general type, and an output variable of a status type (Figure 4). Then, as each input comes in from the bus, it is converted explicitly to the formal parameter type in the argument list. A check is made on the output status when the procedure returns, and processing goes on from there. The status check procedure does not write to any input data, so corruption is eliminated there (guaranteed by using mode IN), and the original input is not damaged or corrupted, because the Ada compiler uses local registers to transfer data from the calling program to the procedure. (NOTE : check your Ada compiler to verify that it uses local registers for parameter passing.) By explicitly converting the data in the argument list we have essentially cheated the software from any opportunity to corrupt the data. The procedure returns the status to the calling unit via an OUT parameter, which is also a local register used by the compiler. (Again, verify with your compiler.)

The status check procedure should avoid the use of IN OUT parameters because the specific implementation of the compiler cannot be determined. A given compiler may use local registers, or may use the actual data memory location for both reads and writes. Using mode IN OUT is also dangerous because Ada guarantees a write to the output data, whether or not it was actually assigned a value. This is also true for mode OUT, but since in this case the OUT data is not related to the IN data, no corruption of the input can occur. (However, the returned output data may not be valid if the OUT data was not assigned a value !)

The combination of data abstraction and data coupling is the key to success in avoiding data corruption. Data abstraction is in this case unit functional typing of data, separate from data assignment, and proper use of WITH and USE for the types and data packages. Data coupling is using parameter lists to share data between subunits rather than global data packages, and avoiding the use of mode IN OUT. Even when the data is of different types, which it often is, similar operations on the data can be performed by using abstraction and coupling to eliminate data corruption.

Data corruption is a serious threat to deliverable, critical software, as evidenced by the FAA's scrutiny of the methods used to avoid it. Ada offers us a combination of ways to operate on the data while shouldering the burden of proof. In addition, Ada allows the best use of its features without putting constraints on the processor or physical environment. As such, it becomes a valuable and desirable language for use in large, complex, and/or critical software projects.

```
TYPE sometype IS RECORD
   data : some_data_type;
   status : bus_validity_type
END RECORD
```

Figure 1 — typical record structure showing data and bus status

```
PACKAGE Common_Deg_C_Types is
—
— PURPOSE : Declare common fixed point types for degrees Celsius types
—
— Status field state values
  TYPE status_type is (BUS_FAIL, NOT_PRESENT, INVALID, VALID);

—Scale 0 Type
Delta_0 : CONSTANT := 1.0/2.0**0;
TYPE Deg_C_Ls0_Type is DELTA Delta_0 RANGE –2.0**31 .. 2.0**31 – Delta_0;
FOR Ls0_Type'small use Delta_0;
.
.
.
— Scale 31 Type
Delta_31 : CONSTANT := 1.0/2.0**31;
TYPE Deg_C_Ls31_Type is DELTA Delta_31 RANGE –2.0**0 .. 2.0**0 – Delta_31;
FOR Ls31_Type'small use Delta_31;
—
— Now generate the actual record types to be used
—
TYPE Deg_C_Ls0_Record_Type is RECORD
   Data : Deg_C_Ls0_Type;
   Status : Status_Type;
END RECORD;
.
.
.
TYPE Deg_C_Ls31_Record_Type is RECORD
   Data : Deg_C_Ls31_Type;
   Status : Status_Type;
END RECORD;

END Common_Deg_C_Types;
```

Figure 2 — Example of types package specification

```
WITH Common_Deg_C_Types;

PACKAGE Egt_Data is
—
— Purpose : Contains all application variable declarations required for Exhaust Gas Temperature Data.
—

— EGT variables

Engine_1_Egt : Common_Deg_C_Types.Deg_C_Ls0_Record_Type;
Engine_2_Egt : Common_Deg_C_Types.Deg_C_Ls0_Record_Type;
Engine_3_Egt : Common_Deg_C_Types.Deg_C_Ls0_Record_Type;
Engine_4_Egt : Common_Deg_C_Types.Deg_C_Ls0_Record_Type;

END Egt_Data;
```

Figure 3 — Example of data package specification

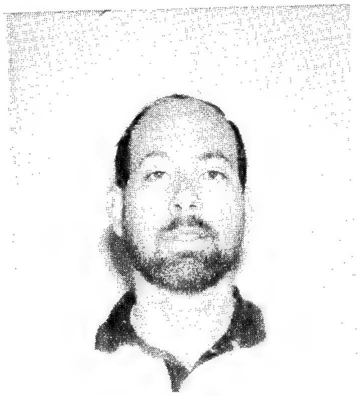
```
PROCEDURE Status_Check ( Data   : IN Common_Deg_C_Types.Deg_C_Ls16_Type;
                         Status : IN Common_Deg_C_Types.Status_Type;
                         Result  : OUT Boolean ) is
—
— PURPOSE : This general procedure takes any variable's data and status as inputs, checks the
—            status for validity and the data for non–zero values, and returns a TRUE for valid
—            status and data and FALSE for invalid or zero data.
—
BEGIN
  IF ( Status /= Common_Deg_C_Types.Valid ) THEN
    Result := False;
   ELSIF ( Data = 0 ) THEN
     Result := False;
   ELSE
      Result := True;
   END IF;
```

The call to this procedure looks like this :

```
Status_Check ( Common_Deg_C_Types.Deg_C_Ls16_Type( Egt_Data.Engine_1_Egt.Data )
               Egt_Data.Engine_1_Egt.Status,
               Result );
IF ( Result ) THEN
— proceed with displaying valid data
ELSE
— do not display invalid data
END IF;
```

Figure 4 — Example of general status check procedure and the call to the procedure

Mike Jenkins is a Principal Engineer in the EFIS Displays department of Honeywell. He is currently the technical cognizant software engineer in the Maintenance and Synoptics group. He has previously worked in flight controls software and simulation for Honeywell, GE, Sperry, and Northrop. Mike holds a B.S. in Aerospace Engineering from the University of Arizona. He can be reached at :
jenkins@saifr00.ateng.az.honeywell.com, or
Honeywell, M/S 2K35C1
POB 21111
Phoenix, AZ 85036–1111



THE ROLE OF THE ADA COMPILER EVALUATION SYSTEM IN SELECTION AND USE OF A COMPILATION AND EXECUTION ENVIRONMENT

Phil Brashear
CTA INCORPORATED
Dayton, Ohio

In this paper, we highlight the importance of basing compilation system selection on sound, repeatable data that can be used to compare performance and usability features of different systems. We discuss the importance of obtaining such data (the "evaluation" process) and provide a quick look at some of the methods of doing so. Then we use the Ada Compiler Evaluation System (ACES) to illustrate the evaluation process and reporting of results.

The Importance of Evaluation

It is easy to underestimate the impact that the compilation and execution system can have on development cost and schedule. A compilation system that provides only minimal diagnostic information, a symbolic debugger that lacks any support for Ada tasking, or a library system that is easily corrupted can cause enormous frustration and schedule overruns. A code generator that produces inefficient code can force developers to expend great effort on hand-optimizing source code. Either kind of situation can cause whole organizations to take the view that the Ada language is not appropriate to system development. The end result may well be the decision to seek waivers of the Ada requirement on all subsequent projects, moving to languages that do not provide the same degree of support for software engineering and the resulting reliability and maintainability.

In order to avoid such catastrophes, both acquisition and development organizations must understand the importance of basing the selection of implementations on data that can be used to compare the performance and usability of various candidate systems. Moreover, the data used in selection must relate to the specific issues of interest to the organization or project. It does little good to know that an implementation has the fastest task switching and rendezvous times if the organization is producing a transaction processing system with no concurrency requirements. Similarly, high-speed data transfer between memory and disk is not a useful predictor of the ability of a system to meet the timing deadlines required by a cyclic executive for an avionics system.

Another issue is the applicability of evaluation results to different computer systems. Selection of an Ada implementation must be based on data produced by evaluating the actual system to be used by the project. Even if the host computer, target computer, both operating systems, and the compiler version are nominally the same, differences in hardware options, installation parameters, and system organization can have enormous impact on actual efficiency and productivity.

In short, selection of an Ada implementation must be based on sound, repeatable, comparable data. The data must include compile-time performance information, execution-time performance information, and information regarding the usability of the system. Finally, the decision must be based on data that is pertinent to the project's needs and that is produced on the actual development and production hardware systems.

Evaluation Tools

Evaluation data can be obtained in several ways. There are classical benchmarks for execution speed. Various organizations have developed extensive proprietary benchmark suites. The Performance Issues Working Group (PIWG) of the Special Interest Group on Ada (SIGAda) sponsors a set of tests referred to as the PIWG benchmarks. The Ada Joint Program Office (AJPO) has sponsored the development of the Ada Compiler Evaluation Capability (ACEC) and its merger with the United Kingdom's Ada Evaluation System (AES). The ACES is the result of this merger.

Classical benchmarks are intended to provide execution-time measurements that can be compared across systems. For example, the Dhrystone benchmark contains a mix of statements reflecting the observed frequency of statements in system programming[1]. Likewise, the Whetstone benchmark contains a mix of statements typical of scientific computations[2]. The classical benchmarks are widely available, and results obtained by running them on several Ada implementations can be compared. However, no single benchmark can be expected to match the needs of a particular project. Moreover, the classical benchmarks are not intended to measure anything other than execution time; we have already seen that other issues are equally important. Thus, the classical benchmarks are not sufficient as a basis for selecting an Ada implementation.

Various Ada compiler implementors and application development organizations have developed their own suites of performance tests, many of which are quite thorough and provide their owners with valuable performance information. However, such proprietary suites are generally not available to other organizations. Even if the owners publish data that is applicable to a project's needs, such data cannot be compared to that from other candidate implementations, for they will not have been tested with the same suite. Proprietary test suites are well suited to use by the owning organization, but are not of much use to others.

The PIWG test suite has the advantage of being widely available and providing fairly broad coverage of the language. It is easy to use, being simple enough that a novice user can generally load and execute the suite in half a day (on one of the systems for which system-specific timing measurement code has been developed). The PIWG suite includes execution speed tests for many simple language features, along with the Whetstone and Dhrystone benchmarks. The compilation speed tests are not considered to be mature enough for widespread use, and there are no tests for usability (as separate from performance). The Winter 1990 Special Edition of *Ada Letters* contains much valuable information on the PIWG test suite.

The ACES is a more comprehensive evaluation system than those described above. It has the advantage of being widely available (see "Points of Contact") and is designed to be portable to all validated Ada implementations. (Of course, no system can be expected to be 100% portable. The ACES contains some tests for implementation-dependent features, and the support software contains some units that must be modified. For example, if CPU time is desired, rather than elapsed time, then a system-specific routine must be supplied to return the CPU time.) The ACES includes over 1700 tests of run-time performance, measuring both execution speed and code size. These tests range from very small tests of specific language features to classical benchmarks and sample application code drawn from actual applications. Compilation and linking speed (either together or separately) may be reported for more than 650 main programs formed from the individual run-time tests. The ACES also contains four usability assessors (software and scripted scenarios). These assessors provide usability data for the diagnostics system, the symbolic debugger, the library management system, and the run-time and compile-time capacities of the system. The ACES also provides analysis tools for comparing the performance of several implementations. See ("Evaluation Reports") for more detail.

In the remainder of this paper, we focus on the use of the ACES, the kinds of data that it produces, and the interpretation of such data.

Preparation for Evaluation

We earlier stated the need for evaluation data that is pertinent to the organization selecting an implementation. The ACES can produce far more data than an organization would normally need; thus, there should be early identification of the needed data.

The first step in performing an evaluation for the purpose of selecting an implementation must be the establishment of clear goals for the evaluation. Among the decisions that must be made are the following:

1. *What weights should be assigned to compilation and linking time, code size, and execution time?*

   The ACES can produce reports based on five metrics: compilation time, linking time, combined compilation and linking time, code size, and execution time. Their relative importance depends heavily on the organization's development methodologies and on the requirements of the application.

2. *Should time be reported in terms of elapsed (wall clock) time or CPU time?*

   Wall clock time can be measured in a portable fashion (by using the predefined Calendar package). If the evaluator wishes, and can provide an implementation-dependent CPU function, CPU time can be measured. The choice depends on the kind of development and execution environment. For single-user systems, the wall clock time and CPU time should be highly correlated. For multi-user systems, wall-clock time is probably more useful in measuring compilation speed, but CPU time is probably more important for execution speed.

3. *Which areas of the language are most important to the application?*

   The ACES performance tests are organized into 17 groups, including a group consisting of sample application code, a classical benchmarks group, and a group intended to systematically measure compilation speed. Other groups include tests of numerical operations, data structure efficiency, effectiveness of optimization, performance of tasking and generics, and the overhead of storage allocation and deallocation. Some of these groups will be of greater importance than others, depending on the application. The ACES reports a comparative summary measure for each system's performance on each group for each of the five metrics identified in (2) above. When selecting an implementation for a particular project, defining a weight for each metric and each group could be a useful approach.

4. *Are there any absolute performance criteria?*

   A real-time project may dictate minimum acceptable task switching time, minimum acceptable rendezvous time, or minimum achievable specified delay. A project involving lots of file manipulation might require minimum file access times. Such criteria should be established before the first candidate system is evaluated.

5. *Are there minimum usability characteristics?*

   A large project may specify minimum values for the number of units that can be compiled into a library. A project depending heavily on concurrency may require a symbolic debugger to have tasking capabilities. An organization that typically makes frequent "builds" of software may require a minimum compilation or linking speed.

6. *What relative weights are attached to usability criteria?*

   Some rating scale is needed for comparing usability characteristics. The ACES does not provide summary or analysis tools for the results of the four usability assessors, so this scaling may not be simple. It clearly depends on the needs of the project.

Like any other activity, evaluation requires forethought. It is not appropriate to simply start testing, produce all the possible reports, and then try to make a decision in the face of an overwhelming mass of data.

A Scenario for ACES Evaluation

Once the goals of the evaluation have been stated, the actual evaluation can begin. The Pretest and performance testing activities described below are applied to each candidate implementation separately; the analysis step is performed after the performance testing is completed for all candidate systems. The assessors are run on each system, but are independent of the performance testing and analysis steps. Therefore, the following describes only the performance testing and analysis. Extensive detail is given in the ACES User's Guide[4]. The ACES provides an extensive Pretest activity that must be completed for each candidate system. For cross-compilation systems, much of the Pretest activity takes place on the host system, but certain units must be executed on the target. The Pretest activity achieves several goals, as follows:

1. Library units are compiled (and tested) for use by the performance tests and analysis programs;

2. Baseline timing measurements are taken that are used during performance testing to ensure that timing measurements are stable and accurate;

3. The Harness program, used to generate command scripts and to track status of performance tests, is compiled, linked, and tested; and

4. The analysis programs are compiled, linked, and tested.

The second major step in the evaluation of a system is the compilation and execution of the performance tests. The Harness program may be used to interactively track the status of each of the run-time performance tests. It provides the capability of generating command scripts, tailored to the particular implementation, for compiling, linking, and executing selected collections of tests. The scripts are executed, the results are captured in log files, and Harness extracts status data from the logs. This process is repeated until the evaluator is satisfied that the maximum set of test results has been captured. Then a data extraction program is executed, extracting the time and size data from the log files and formatting it for use in analysis.

After all candidate systems have processed all the performance tests, and all the performance data has been formatted, the Comparative Analysis program is executed. The user uses a text-based menu interface to select the particular reports and formats that are desired, and the analysis program provides comparison data for the tested systems. Some of these reports are discussed in the next section.

Table 1
Raw Data

```
                      GENERICS (gn) - product model
=========================================================================
    ---- Raw data
=========================================================================
    Raw Data:                |      sys_1        sys_2        sys_3   |  Weights


-------------------------------------------------------------------------
         instantiation (in)                        -- Missing:  1
    enum_io_01               |      58.10        17.20       595.90   |      1.0
    enum_io_02               |     115.00        36.90      1263.00   |      1.0
    . . .
         subprogram (su)                           -- Missing:  1
    subprogram_01            |   20013.20     40833.20       721.00   |      1.0
    . . .
    subprogram_04            |      21.90         5.89         0.00   |      1.0
    subprogram_05            |                   33.10         1.02   |      1.0
    . . .
    subprogram_15            |                   41.90         3.80   |      1.0
-------------------------------------------------------------------------
         ---- Total missing:   2
=========================================================================
```

Evaluation Reports

In the following paragraphs, the illustrative reports are duplicated or adapted from the ACES Reader's Guide[3]. All information identifying the tested systems has been suppressed.

While the comparative reports produced by the ACES analysis software are quite extensive, three kinds of tables provide the core data. At the user's option, each of the 17 test groups described above has two of these tables for each of the 5 metrics described earlier. Thus, there are 85 pairs of tables, and each set has supplementary data with it. In addition, a set of summary tables is produced for each metric. For consistency in discussion, we focus on the execution time metric.

In each table, the columns represent the systems being compared, and the rows represent the individual test problems (see Table 1).

In this table, three systems are being compared. Execution time results (in microseconds) are presented for the tests in the Generics test group. The indented names ("instantiation" and "subprogram") identify organizational subgroups of the Generics group. Note that, in each subgroup, one test is omitted. No test result is reported unless at least two systems have produced data for that test (otherwise, no comparison is possible). The first tested system reports no results on several of the "subprogram" tests; these tests are still used in the comparison, since the other two systems produced results for them. The "Weights" column gives the relative weight assigned to each test by the user; the default weight is 1.0 for each test.

The other two kinds of analysis table present data in terms of a "product model" in which each raw measurement is factored into three components, as indicated by the first line of Table 2.

In Table 2, the Problem Factors (row means) appear in the rightmost column, and the System Factors (analogous to column means) appear in the bottom row. The body of the table contains the Residuals, with indications that some are outside statistical expectation: either too low ("--") or too high ("++").

Table 2
Residuals and System Factors

```
============================================================================
    ---- Residual * System Factor * Row Mean = Actual
============================================================================
    Residuals:              |     sys_1        sys_2        sys_3    |    Means
----------------------------------------------------------------------------
         instantiation (in)                     -- Missing:  1
    enum_io_01              |      0.28       0.04--      17.58++ |    223.73
    enum_io_02              |      0.26       0.04--      17.67++ |    471.63
     . . .
         subprogram (su)                        -- Missing:  1
    subprogram_01           |      1.05       1.07         0.23   |  20522.47
     . . .
    subprogram_04           |      2.54       0.34         0.00-- |      9.26
    subprogram_05           |                 1.04         0.39   |     17.06
     . . .
    subprogram_15           |                 0.98         1.10   |     22.85
----------------------------------------------------------------------------
         ---- Total missing:    2
============================================================================
    System Factor           |      0.93       1.87         0.15   |
============================================================================
```

The analysis calculates the System Factor values that provide the best fit to the product model. For a perfect fit to the model, each column of results (for one system) would be a scalar multiple of the row (problem) mean. The multiplier is the System Factor. That is, if the data perfectly fit the model, then each raw result would be the product of the System Factor for its column and the row mean (Problem Factor) for its row. All residuals would be 1.0.

Thus, the System Factors are measures of central tendency, representing the system's contribution to the raw result. Likewise, the Problem Factors are measures of central tendency, representing the problem's contribution to the raw result. The Residuals indicate how far the particular raw measure deviates from the product model. The Problem Factors are row means; calculation of the System Factor that provides the best fit to the product model requires more sophisticated statistical calculations.

The product model provides summary values that can be plausibly (though not precisely) used in such statements as "System sys_2, on the average, takes more than 12 times as long to execute tests in the Generics group as does system sys_3." (The ratio of the System Factors is 1.87 / 0.15, or about 12.47.) Conversely, we could make the equivalent statement that system sys_3 is 12.47 times as fast as system sys_2.

The third kind of table appears only once for each metric, in the Summary of All Groups section. This is actually a Raw Data table, as was illustrated in Table 1, but the values used for the raw data are the System Factors of the tested systems for the 17 test groups. These five summary tables provide a convenient quick-look summary of the performance of all the systems on all the groups, and may provide all the performance data needed for selection. However, there are related tables that give confidence intervals and indications of significant differences. It would be wise to examine these, along with the associated residual tables, before concluding that the differences in System Factors are large enough to affect the selection decision.

Table 3
Summary of All Groups

```
=========================================================================
Raw Data:                |        sys_1        sys_2        sys_3 |  Weights
-------------------------------------------------------------------------
application              |         3.05         0.26         1.31 |      1.0
arithmetic               |         2.06         0.25         1.22 |      1.0
 . . .
systematic_compile_speed |         3.81         0.26              |      1.0
tasking                  |         2.25         0.51         0.36 |      1.0
=========================================================================
```

Summary

We have seen the necessity of supporting Ada compilation system selection with hard data that can be replicated and can be compared across systems. Several popular sources of evaluation data were briefly described, along with mention of their advantages and disadvantages. The Ada Compiler Evaluation System (ACES) was used to illustrate the major steps in the evaluation process, and examples of the most important ACES comparison reports were provided.

The final point to be emphasized is that trustworthy, widely useful performance data can be generated, and should always be used when performance and usability are to be cited for any reason, especially for selecting an Ada compilation and execution system.

Points of Contact

The ACES software and machine-readable documentation are available by anonymous FTP from the AJPO host (*ajpo.sei.cmu.edu*), in subdirectory *public/aces.* The subdirectory contains a README file that describes the contents. There are six files, each of which results from applying the UNIX *compress* utility to a UNIX *tar* file.

The Data Analysis Center for Software distributes ACES for a small fee. The software and machine-readable documentation are provided on a single VMS/BACKUP reel tape, and printed copies of the documentation are included. Send inquiries to:

Data Analysis Center for Software
P.O. Box 120
Utica, NY 13503
(315) 734-3696

For further information, contact the author or the Government Contract Officer's Technical Representative (COTR) as follows:

Dale Lange
645 CCSG/SCSL
3810 Communications, Suite 1
Wright-Patterson AFB, OH 45433
(513) 255-4472
langed@adawc.wpafb.af.mil

References

1. Weicker, R.P., "Dhrystone Benchmark (Ada Version 2): Rationale and Measurement Rules", Ada Letters Vol. IX, Number 5 (July/August 1989)

2. Clapp, R.M. and T. Mudge, "Introduction" in "A Rationale for the Design and Implementation of Ada Benchmark Programs", Ada Letters Vol. X, Number 3 (Special Edition, Winter 1990)

3. *Ada Compiler Evaluation System Reader's Guide*, U.S. Air Force (1993)

4. *Ada Compiler Evaluation System User's Guide*, U.S. Air Force (1993)

Author Information

Phil Brashear
CTA INCORPORATED
5100 Springfield Pike, Suite 100
Dayton, OH 45431
(513) 258-0831
brashear@ajpo.sec.cmu.edu

Mr. Brashear is the ACES technical manager on CTA's contract to support the High Order Language Facility. He has been involved in Ada compiler validation and evaluation since 1986. Before entering the Ada contractor environment, he was Associate Professor of Computer Science at Eastern Kentucky University. In this capacity, he was one of the earliest to teach Ada in a university environment.

Implementation of Six Object-Oriented Metrics

Bill Fay
u7c31@gsde.hso.link.com

Viktor Ohnjec
vo@rational.com

ABSTRACT

Metrics help software professionals estimate the cost, schedule and manpower requirements of current and future projects, evaluate the productivity impacts of new tools and techniques, establish productivity trends over time, improve software quality and anticipate and reduce future maintenance costs.

However, software metrics used to measure traditional software development products do not consider the object-oriented notions of classes, inheritance, encapsulation and message passing. Because of this, Chidamber and Kemerer have proposed six metrics specifically intended to measure object-oriented software. These six metrics are:

- o Weighted Methods Per Class
- o Depth of Inheritance Tree
- o Number Of Children
- o Coupling Between Objects
- o Response For A Class
- o Lack Of Cohesion In Methods

This paper provides definitions and rationale for each of these metrics along with examples derived from a flight vehicle hydraulic system simulated by a real-time software system. The implementation is object-oriented and was produced using the Ada programming language.

Several software engineering principles are illustrated by including them in the design and implementation of the example. These principles include using and creating reusable components, multi-level abstractions, information hiding, and modularity.

This paper also shows how Ada is used to implement the object-oriented concepts of inheritance, aggregation, polymorphism and encapsulation.

BACKGROUND

The Space Station Verification and Training facility (SSVTF) is a software-intensive application utilizing object-oriented development techniques and is being implemented in Ada. Since it is a NASA program, contractors must maintain several NASA program requirements, one of which states that certain software metrics will be collected and reported such that the status of the project may be monitored by NASA personnel. Since the SSVTF program was among the first major object-oriented programs administered by NASA, the contractors had to ensure that the metrics reported were meaningful in the context of various other software development methodologies. Upon review of NASA-mandated metrics, it was realized that "traditional" metrics such as McCabe Complexity and number of Ada instantiations, by themselves, were without meaning for the object-oriented implementation of the SSVTF.

At OOPSLA '91, Shyam Chidamber and Chris Kemerer[1] presented a paper on six object oriented metrics. These six metrics, which are currently under review for use on the SSVTF, are summarized here. The application of these metrics to an object-oriented implementation of a simulated hydraulic pressurization system is also provided to show how the metrics are applied to an Ada implementation.

WEIGHTED METHODS PER CLASS

Definition:

For a class with n methods m_1, m_2...m_n each with a complexity measure $c_{1,}$ c_2...c_n then the WMC is c1 + c_2 + ... + c_n.

Rationale:

The number of methods and their corresponding complexity indicate the amount of effort required to develop and maintain the class.

If we accept that McCabe's Complexity Measure[2] provides an indication of the amount of effort required to develop and maintain a software module (i.e. modules with high measures require more effort than modules with low measures) then we may argue that WMC provides insight to the effort required to develop and maintain an entire class.

Since a child class inherits all the methods of its parent, the larger the number of methods in a class, the greater the potential impact on children. Suppose that a (parent) class has a high WMC measure and that class is inherited by a (child) class. Then, the amount of effort required to comprehend the child is greater than if the parent has a low WMC measure.

Classes with large numbers of methods, or classes with methods having high complexity measures are likely to be more application specific, limiting the possibility of reuse. In addition, as the WMC measure increases, the reusability of a class decreases. This is supported by the concept of inheritance; a child class inherits the parent class and becomes less general. Thus, as in inheritance, as the number of methods in a class and/or the complexity of methods increase, the generality of the class decreases, limiting reuse.

DEPTH OF INHERITANCE TREE

Definition:

The height of the class in the inheritance tree (i.e. the length, in number of nodes, of the longest path from the node to the root of the tree).

Rationale:

The deeper a class is in an inheritance hierarchy, the number of methods it inherits necessarily increases. Provided that the complexity of the methods of two classes are roughly equivalent, the class with a high number of methods is likely to be more difficult to comprehend than a class with fewer methods. Deeper inheritance trees constitute greater design complexity since more methods and classes are involved.

The deeper a particular class is in the inheritance hierarchy, the greater the potential reuse of inherited methods. In other words, the most general class is the class that is the deepest in the inheritance hierarchy; all classes above inherit this class and become more specific. And, the most general class is also the most reusable.

NUMBER OF CHILDREN

Definition:

Number of immediate subclasses subordinated to a class in the class hierarchy.

Rationale:

NOC provides a measure of scope of properties for a class since it measures the number of subordinate classes that inherit the parent's methods.

As the number of children for a class increases, reuse also increases since inheritance is the mechanism for reuse. Thus, NOC indicates reuse.

The greater the number of children of a class, the greater the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of inheritance. Therefore, a class with a high NOC measure should be verified that it is not overly general.

The number of children for a class indicates the influence a class has on the design. If a class has a large number of children, it may require more thorough testing of the methods in that class. Effort expended in testing and debugging of a class that has a relatively high NOC measure has a potentially greater return than effort spent on a class with a low NOC measure.

COUPLING BETWEEN OBJECTS

Definition:

CBO for a class is the number of couples with other classes (i.e. the number of classes on which the class depends)

Rationale:

Excessive coupling between objects is detrimental to modular design and prevents reuse. The more independent an object is, the easier it is to reuse it in another application.

In order to improve modularity and promote encapsulation, inter-class couples should be minimized. The larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult.

A measure of coupling is useful to determine how complex the testing of various parts of a design are likely to be. The higher the inter-class coupling, the more rigorous the testing needs to be.

RESPONSE FOR A CLASS

Definition:

RFC: The sum of the number of methods defined in a class and the number of methods (both internal and external) called by those methods.

Rationale:

This is a measure of all methods in a class and the methods called by those methods. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester.

The larger the number of methods that can be invoked from a class, the greater the complexity of the class. A worst case value for possible responses will assist in appropriate allocation of testing time.

LACK OF COHESION IN METHODS

Definition:

Given a class with methods M_1, M_2 ..., M_n, let $\{I_i\}$ = set of instance variables used by method M_i. LCOM is the number of disjoint sets formed by the intersection of the n sets.

Rationale:

Cohesiveness of methods within a class is desirable, since it promotes encapsulation. Conversely, lack of cohesion implies that classes should probably be split into two or more subclasses. Furthermore, low cohesion increases complexity, thereby increasing the likelihood of errors during the development process. Therefore, any measure of desperateness of methods helps identify flaws in the design of classes.

EMPIRICAL DATA

The following is a tabular listing of actual measures of these six metrics as taken from the pressurization system implementation given in Appendix A.

An interesting observation is that the Coupling Between Objects (CBO) and Lack of Cohesion in Methods (LCOM) measures are low. The implementation methodology enforces these low measures since each class typically defines only three methods; 'Create', 'Periodic_Update' and 'Modify'.

HYDRAULIC SYSTEM OVERVIEW

A real-world flight vehicle hydraulic system provides pressurized hydraulic fluid to actuators that move the control surfaces and raise and lower the landing gear. The hydraulic system includes two fluid pressurization assemblies that each include one motor, one gear box and one pump, two valves, two accumulators, one reservoir, one reservoir quantity sensor, two pressure sensors, a fluid distribution system, fluid return lines and a fluid pressurization assembly

The hydraulic system interacts with two major external systems; the hydraulic control panel and the electrical system. The hydraulic control panel has switches and indicators that control and display the hydraulic system status. The electrical system provides power to the components in the hydraulic system which in turn place loads upon the electrical system.

For the purposes of this paper, only the fluid pressurization assembly will be shown. A class relationship diagram of the pressurization assembly is shown in Figure 1.

WMC	DIT	NOC	CBO	RFC	LCOM	Class Name
13	1	1	0	3	2	Actuator_Class
8	2	1	1	6	2	Axial_Piston_Pump_Class
3	1	1	0	2	2	Centrifugal_Pump_Class
9	1	1	0	3	2	Dc_Motor_Class
7	2	1	2	6	0	Drive_Unit_Class
6	1	1	0	3	2	Gear_Box_Class
11	3	1	3	11	2	Hydraulic_Pump_Class
5	1	1	0	3	0	Positive_Displacement_Pump_Class
9	4	0	2	17	1	Pressurization System Class

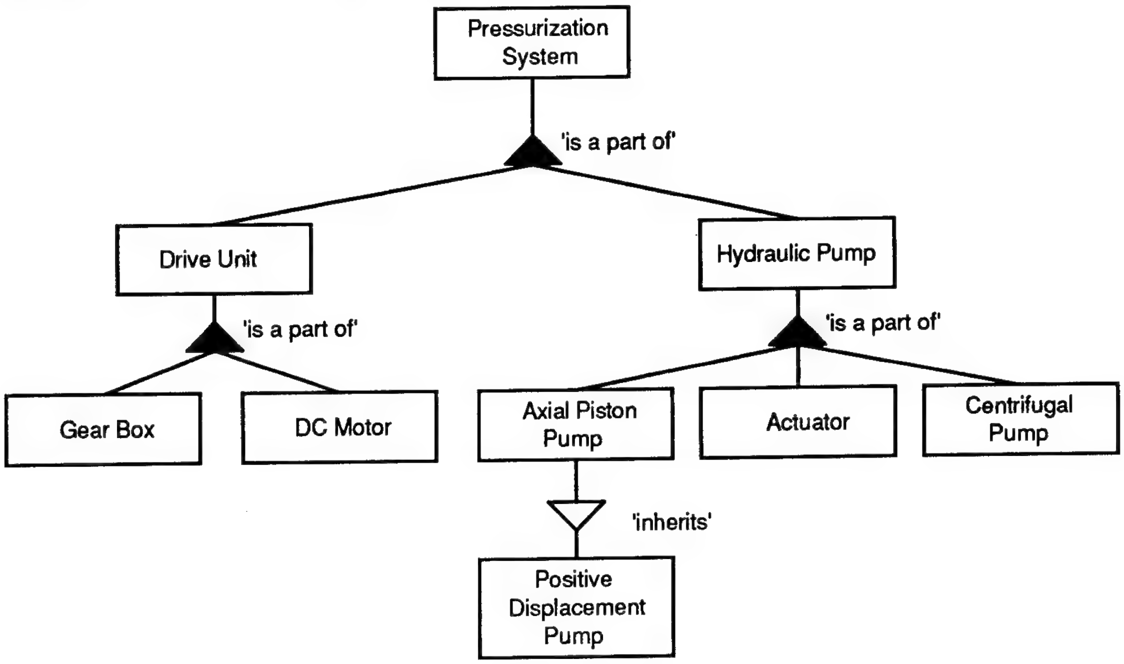


Figure 1: Pressurization System Class Relationship Diagram

ADA IMPLEMENTATION

Classes

Classes are implemented as Ada packages and encapsulate operations (modifiers and selectors) and attributes.

Attributes

Attributes define the state of an instance of a class. Class attributes are collected in a single Ada record type and are made unavailable outside of the package by declaring the record as 'limited private'. This allows an object to be declared of the record type, without providing external visibility to the object's state. Instead, the attribute values are accessed via modifiers and selectors.

Modifiers

Modifiers are Ada procedures defined in a class package that allow the state of the object to be changed. Modifier names are in the form of an action verb. The specific instance of the class (the Ada data object) is passed to the procedure as an 'in out' parameter which allows the modifier to have access to all attributes defined for the class and also allows the modifier to change any attributes of the class (hence the name 'modifier').

Selectors

Selectors are Ada functions defined in a class package that return the current value of an attribute associated with an object. Selector names are in the form of a noun. The name 'selector' implies that the function provides the attribute value but does not allow the value to be changed.

Commands

Commands and associated data are defined for each class, allowing a single operation to be used to modify the data associated with the attributes of a class. These commands are implemented in the form of a discriminated (variant) record type which is defined to have a polymorphic structure[3]. That is, a single modifier is defined that allows all sets of state variables associated with the commands to be updated.

Appendix 1 - Pressurization System Source Code

The following is the Ada source code that implements the pressurization system shown in Figure 1. The code is 100% semantically correct (i.e. it is compilable).

actuator_class specification

```
with Common_Types;

package Actuator_Class is

  type Object is limited private;

  package Ct renames Common_Types;

  -- ********************* Commands **********************

  type Command_Kinds is (Common_Leak);

  type Commands (Kind : Command_Kinds := Common_Leak) is
    record
      case Kind is
        when Common_Leak =>
          Leak_Rate : Ct.Gallons_Per_Second := 0.0;
      end case;
    end record;

  -- ********************* Modifiers ********************

  procedure Create (Instance : in out Object);

  procedure Modify (Instance : in out Object;
                    Command : in Commands);

  procedure Periodic_Update (Instance : in out Object;
                             Delta_Time : in Ct.Seconds;
                             Presaure : in Ct.Psi);

  -- ********************* Selectors ********************

  function Stroke (Instance : in Object) return Ct.Feet;

  -- ****************************************************

private

  type Attributes;

  type Object is access Attributes;

end Actuator_Class;
```

actuator_class body

```
with Common_Types;
use Common_Types;

package body Actuator_Class is

  type Inches_Per_Second_Squared is new Ct.Sp_Real;

  -- ********************* Attributes ********************

  type Attributes is
    record
      Leak_Rate : Ct.Gallons_Per_Second := 0.0;
      Spool_Force : Ct.Foot_Pounds := 0.0;
      Friction : Ct.Foot_Pounds := 0.0;
      Spring_Force : Ct.Foot_Pounds := 0.0;
      Accel : Inches_Per_Second_Squared := 0.0;
      Velocity : Ct.Feet_Per_Second := 0.0;
      Position_Lim : Ct.Feet := 0.0;
      Swash_Plate_Angle : Ct.Radians := 0.0;
      Stroke : Ct.Feet := 0.0;
    end record;

  -- ********************* Modifiers ********************

  procedure Create (Instance : in out Object) is
begin
  Instance := new Attributes;
end Create;


procedure Calculate_Spool_Force (Instance : in out Object) is
begin
  -- NOTE: Operational details have been omitted.
  null;
end Calculate_Spool_Force;


procedure Calculate_Accel (Instance : in out Object) is
begin
  -- NOTE: Operational details have been omitted.
  null;
end Calculate_Accel;


procedure Calculate_Friction
            (Instance : in out Object; Press_Input : Ct.Psi) is
begin
  -- NOTE: Operational details have been omitted.
  null;
end Calculate_Friction;


procedure Calculate_Velocity (Instance : in out Object) is
begin
  -- NOTE: Operational details have been omitted.
  null;
end Calculate_Velocity;


procedure Calculate_Position (Instance : in out Object) is
begin
  -- NOTE: Operational details have been omitted.
  null;
end Calculate_Position;


procedure Calculate_Swash_Plate_Angle
            (Instance : in out Object) is
begin
  -- NOTE: Operational details have been omitted.
  null;
end Calculate_Swash_Plate_Angle;


procedure Calculate_Stroke (Instance : in out Object) is
begin
  -- NOTE: Operational details have been omitted.
  null;
end Calculate_Stroke;


procedure Calculate_Adjustment_Spring
            (Instance : in out Object) is
begin
  -- NOTE: Operational details have been omitted.
  null;
end Calculate_Adjustment_Spring;


procedure Modify (Instance : in out Object;
                  Command : in Commands) is
begin
  case Command.Kind is
    when Common_Leak =>
      Instance.Leak_Rate := Command.Leak_Rate;
  end case;
end Modify;


procedure Periodic_Update (Instance : in out Object;
                           Delta_Time : in Ct.Seconds;
                           Pressure : in Ct.Psi) is

  Total_Force : Ct.Foot_Pounds;

begin

  Calculate_Friction (Instance => Instance,
                      Press_Input => Pressure);
    Calculate_Adjustment_Spring (Instance => Instance);

    Calculate_Spool_Force (Instance => Instance);

    Total_Force := Instance.Spool_Force -
                     Instance.Spring_Force - Instance.Friction;

    Calculate_Accel (Instance => Instance);

    Calculate_Velocity (Instance => Instance);

    Calculate_Position (Instance => Instance);

    Calculate_Swash_Plate_Angle (Instance => Instance);

    Calculate_Stroke (Instance => Instance);

  end Periodic_Update;

  -- ******************** Selectors ********************

  function Stroke (Instance : in Object) return Ct.Feet is
  begin
    return (Instance.Stroke);
  end Stroke;

end Actuator_Class;
```

axial_piston_pump_class specification

```
with Common_Types;

package Axial_Piston_Pump_Class is

  type Object is limited private;

  package Ct renames Common_Types;

  -- ********************* Commands *********************

  type Command_Kinds is (Set_Efficiency, Set_Delta_Flow);

  type Commands (Kind : Command_Kinds := Set_Efficiency) is
    record
      case Kind is
        when Set_Efficiency =>
          Efficiency : Ct.Non_Dimensional := 1.0;
        when Set_Delta_Flow =>
          Delta_Flow : Ct.Gallons_Per_Second := 0.0;
      end case;
    end record;

  -- ********************* Modifiers ********************

  procedure Create (Instance : in out Object;
                    Displacement : in Ct.Gallons;
                    Efficiency : in Ct.Non_Dimensional;
                    Number_Of_Pistons : in Natural;
                    Piston_Area : in Ct.Square_Feet);

  procedure Modify (Instance : in out Object;
                    Command : in Commands);

  procedure Periodic_Update (Instance : in out Object;
                             Pressure : in Ct.Psi;
                             Stroke : in Ct.Feet;
                             Rotation : in Ct.Radians_Per_Second);

  -- ********************* Selectors ********************

  function Flow (Instance : in Object)
              return Ct.Gallons_Per_Second;

  function Pressure (Instance : in Object) return Ct.Psi;

  function Torque (Instance : in Object) return Ct.Foot_Pounds;

  function Efficiency (Instance : in Object)
                    return Ct.Non_Dimensional;

  -- *****************************************************

private

  type Attributes;

  type Object is access Attributes;

end Axial_Piston_Pump_Class;
```

axial_piston_pump_class body

```
with Positive_Displacement_Pump_Class;

with Common_Types;
use Common_Types;

package body Axial_Piston_Pump_Class is

  -- ******************** Attributes ********************

  type Positive_Displacement_Pump is
    new Positive_Displacement_Pump_Class.Object;

  type Attributes is
    record
      Pd_Pump : Positive_Displacement_Pump;
      Pressure : Ct.Psi := 0.0;
      Scale : Ct.Non_Dimensional := 0.0;
      Bias : Ct.Gallons_Per_Second := 0.0;
      Torque : Ct.Foot_Pounds := 0.0;
      Flow : Ct.Gallons_Per_Second := 0.0;
      Delta_Flow : Ct.Gallons_Per_Second := 0.0;
      Loss_Flow : Ct.Gallons_Per_Second := 0.0;
    end record;

  -- ******************** Modifiers ********************

  procedure Create (Instance : in out Object;
                    Displacement : in Ct.Gallons;
                    Efficiency : in Ct.Non_Dimensional;
                    Number_Of_Pistons : in Natural;
                    Piston_Area : in Ct.Square_Feet) is

    Pd_Pump : Positive_Displacement_Pump;

    function Initial_Flow (Displacement : in Ct.Gallons)
                          return Ct.Gallons_Per_Second is
    begin
      return Gallons_Per_Second (Displacement * 3.4231);
    end Initial_Flow;

  begin

    -- Create the Axial Piston Pump
    Instance := new Attributes;

    Instance.Pressure := 60.0;
    Instance.Scale := 1.0;
    Instance.Bias := 0.0;
    Instance.Torque := 121.0;
    Instance.Flow := Initial_Flow (Displacement);
    Instance.Delta_Flow := 0.0;
    Instance.Loss_Flow := 0.0248;

    -- Create the Positive_Displacement_Pump
    Create (Instance => Instance.Pd_Pump,
            Displacement => Displacement,
            Efficiency => Efficiency);

  end Create;


  procedure Modify (Instance : in out Object;
                    Command : in Commands) is

  begin
    case Command.Kind is

      when Set_Efficiency =>
        declare
          Pdpump_Command :
            constant Positive_Displacement_Pump_Class.Commands :=
            (Kind => Positive_Displacement_Pump_Class.
                       Set_Efficiency,
             Efficiency => Command.Efficiency);
        begin
          -- Modify the Positive_Displacement_Pump
          Modify (Instance => Instance.all.Pd_Pump,
                  Command => Pdpump_Command);
        end;

      when Set_Delta_Flow =>
        Instance.Delta_Flow := Command.Delta_Flow;
    end case;

  end Modify;
```

```
procedure Periodic_Update
            (Instance : in out Object;
             Pressure : in Ct.Psi;
             Stroke : in Ct.Feet;
             Rotation : in Ct.Radians_Per_Second) is

  function Efficiency (Pressure : in Ct.Psi;
                       Rotation : in Ct.Radians_Per_Second)
                      return Ct.Non_Dimensional is
    -- a local function
  begin
    return 1.0;
  end Efficiency;

begin

  -- Update the Positive_Displacement_Pump
  Periodic_Update (Instance => Instance.all.Pd_Pump,
                   Efficiency => Efficiency
                                   (Pressure, Rotation));

end Periodic_Update;

-- ********************* Selectors *********************

function Flow (Instance : in Object)
              return Ct.Gallons_Per_Second is
begin
  return Instance.Flow;
end Flow;


function Pressure (Instance : in Object) return Ct.Psi is
begin
  return Instance.Pressure;
end Pressure;


function Torque (Instance : in Object) return Ct.Foot_Pounds is
begin
  return Instance.Torque;
end Torque;


function Efficiency (Instance : in Object)
                    return Ct.Non_Dimensional is
begin
  return Efficiency (Instance.Pd_Pump);
end Efficiency;


end Axial_Piston_Pump_Class;
```

centrifugal_pump_class specification

```
with Common_Types;

package Centrifugal_Pump_Class is

  type Object is limited private;

  package Ct renames Common_Types;

  -- ********************* Modifiers *********************

  procedure Create (Instance : in out Object);

  procedure Periodic_Update
            (Instance : in out Object;
             Delta_Time : in Ct.Seconds;
             Supply_Speed : in Ct.Radians_Per_Second;
             Fluid_Avail : in Boolean;
             Consumed_Flow : in Ct.Gallons_Per_Second);

  -- ********************* Selectors *********************

  function Torque (Instance : in Object) return Ct.Foot_Pounds;

  function Pressure (Instance : Object) return Ct.Psi;

  function Consumed_Flow (Instance : Object)
                         return Ct.Gallons_Per_Second;

  -- *****************************************************

private

  type Attributes;

  type Object is access Attributes;

end Centrifugal_Pump_Class;
```

centrifugal_pump_class body

```
package body Centrifugal_Pump_Class is

  -- ********************* Attributes *********************

  type Attributes is
    record
      Torque : Ct.Foot_Pounds := 0.0;
      Press : Ct.Psi := 0.0;
      Flow : Ct.Gallons_Per_Second := 0.0;
    end record;

  -- ********************* Modifiers *********************

  procedure Create (Instance : in out Object) is
  begin
    Instance := new Attributes;
  end Create;


  procedure Periodic_Update
              (Instance : in out Object;
               Delta_Time : in Ct.Seconds;
               Supply_Speed : in Ct.Radians_Per_Second;
               Fluid_Avail : in Boolean;
               Consumed_Flow : in Ct.Gallons_Per_Second) is
  begin
    -- Common flow rate consumed for output
    Instance.Flow := Consumed_Flow;

    -- Function needed to convert supply speed, fluid
    -- availability and delta time into pressure.
    Instance.Press := 45.0;

    -- Function needed to convert supply speed, fluid
    -- availability and delta time into torque.
    Instance.Torque := 645.0;

  end Periodic_Update;

  -- ********************* Selectors *********************

  function Torque (Instance : in Object) return Ct.Foot_Pounds is
  begin
    return (Instance.Torque);
  end Torque;


  function Pressure (Instance : Object) return Ct.Psi is
  begin
    return (Instance.Press);
  end Pressure;


  function Consumed_Flow (Instance : Object)
                         return Ct.Gallons_Per_Second is
  begin
    return (Instance.Flow);
  end Consumed_Flow;

end Centrifugal_Pump_Class;
```

common_types specification

```
with Numeric_Types;

package Common_Types is

  subtype Sp_Real is Numeric_Types.Sp_Real;

  type Amps is new Sp_Real;
  type Feet is new Sp_Real;
  type Feet_Per_Second is new Sp_Real;
  type Foot_Pounds is new Sp_Real;
  type Gallons is new Sp_Real;
  type Gallons_Per_Second is new Sp_Real;
  type Non_Dimensional is new Sp_Real;
  type Psi is new Sp_Real;
  type Radians is new Sp_Real;
  type Radians_Per_Second is new Sp_Real;
  type Seconds is new Sp_Real;
  type Square_Feet is new Sp_Real;
  type Volts is new Sp_Real;
  type On_Off is (On, Off);

end Common_Types;
```

dc_motor_class specification

```
with Common_Types;

package Dc_Motor_Class is

  type Object is limited private;

  package Ct renames Common_Types;

  -- ******************** Commands *********************

  type Command_Kinds is (Motor_Fail);

  type Commands (Kind : Command_Kinds := Motor_Fail) is
    record
      case Kind is
        when Motor_Fail =>
          Apply : Boolean := False;
      end case;
    end record;


  -- ******************** Modifiers ********************

  procedure Create (Instance : in out Object;
                    Max_Voltage : in Ct.Volts;
                    Min_Voltage : in Ct.Volts;
                    Nominal_Load : in Ct.Amps;
                    Nominal_Speed : in Ct.Radians_Per_Second;
                    Nominal_Torque : in Ct.Foot_Pounds);

  procedure Modify (Instance : in out Object;
                    Command : in Commands);

  procedure Periodic_Update (Instance : in out Object;
                             Delta_Time : in Ct.Seconds;
                             Torque : in Ct.Foot_Pounds;
                             Avail_Power : in Ct.Volts);

  -- ********************* Selectors ****************** --

  function Load (Instance : in Object) return Ct.Amps;

  function Shaft_Output (Instance : in Object)
                        return Ct.Radians_Per_Second;

  -- ****************************************************

private

  type Attributes;

  type Object is access Attributes;

end Dc_Motor_Class;
```

dc_motor_class body

```
with Common_Types;
use Common_Types;

package body Dc_Motor_Class is

  -- ******************** Attributes ******************

  type Attributes is
    record
      Nominal_Speed : Ct.Radians_Per_Second := 0.0;
      Nominal_Torque : Ct.Foot_Pounds := 0.0;
      Shaft_Speed : Ct.Radians_Per_Second := 0.0;
      Shaft_Fail : Boolean := False;
      Elec_Load : Ct.Amps := 0.0;
      Max_Voltage : Ct.Volts := 0.0;
      Min_Voltage : Ct.Volts := 0.0;
      Nominal_Load : Ct.Amps := 0.0;
      Power_On : Boolean := False;
    end record;

  -- ******************** Modifiers ********************

  procedure Create (Instance : in out Object;
                    Max_Voltage : in Ct.Volts;
                    Min_Voltage : in Ct.Volts;
                    Nominal_Load : in Ct.Amps;
                    Nominal_Speed : in Ct.Radians_Per_Second;
                    Nominal_Torque : in Ct.Foot_Pounds) is
  begin

    Instance := new Attributes'(Nominal_Speed => Nominal_Speed,
                                Nominal_Torque => Nominal_Torque,
                                Shaft_Speed => 0.0,
                                Shaft_Fail => False,
                                Elec_Load => 0.0,
                                Max_Voltage => Max_Voltage,
                                Min_Voltage => Min_Voltage,
                                Nominal_Load => Nominal_Load,
                                Power_On => False);
  end Create;


  procedure Modify (Instance : in out Object;
                    Command : in Commands) is
  begin
    case Command.Kind is
      when Motor_Fail =>
        Instance.Shaft_Fail := True;
    end case;
  end Modify;


  procedure Periodic_Update (Instance : in out Object;
                             Delta_Time : in Ct.Seconds;
                             Torque : in Ct.Foot_Pounds;
                             Avail_Power : in Ct.Volts) is

  begin

    -- Motor is operational when power is: (min volts <=
    -- avail_power <= max volts)
    Instance.Power_On := (Instance.Min_Voltage <= Avail_Power) and
                           (Avail_Power <= Instance.Max_Voltage);

    -- Based on torque load, available power and
    -- shaft status, determine shaft speed
    if (Torque <= Instance.Nominal_Torque) and
       Instance.Power_On and (not Instance.Shaft_Fail) then

      Instance.Shaft_Speed :=
        Instance.Nominal_Speed *
          Ct.Radians_Per_Second
            ((1.0 - Ct.Non_Dimensional
                      (Torque / Instance.Nominal_Torque)));
    else
      Instance.Shaft_Speed := 0.0;
    end if;

    -- Return constant load if powered, otherwise return 0.0
    if Instance.Power_On then
      Instance.Elec_Load := Instance.Nominal_Load;
    else
      Instance.Elec_Load := 0.0;
    end if;

  end Periodic_Update;

  -- ******************** Selectors ********************

  function Load (Instance : in Object) return Ct.Amps is
  begin
    return Instance.Elec_Load;
  end Load;


  function Shaft_Output (Instance : in Object)
                        return Ct.Radians_Per_Second is
  begin
    return Instance.Shaft_Speed;
  end Shaft_Output;

end Dc_Motor_Class;
```

drive_unit_class specification

```
with Common_Types;

package Drive_Unit_Class is

  type Object is limited private;

  package Ct renames Common_Types;

  -- ******************** Commands *********************
  type Command_Kinds is (Gearbox_Seizure, Motor_Fail);

  type Commands (Kind : Command_Kinds := Gearbox_Seizure) is
    record
      Apply : Boolean := False;
      case Kind is
        when Gearbox_Seizure =>
          null;
        when Motor_Fail =>
          null;
      end case;
    end record;

  -- ********************* Modifiers *********************

  procedure Create (Instance : in out Object;
                    Gearbox_Max_Torque : in Ct.Foot_Pounds);

  procedure Modify (Instance : in out Object;
                    Command : in Commands);

  procedure Periodic_Update (Instance : in out Object;
                             Avail_Power : in Ct.Volts;
                             Delta_Time : in Ct.Seconds;
                             Torque : in Ct.Foot_Pounds);

  -- ********************* Selectors *********************

  function Elec_Load (Instance : in Object) return Ct.Amps;

  function Motor_On (Instance : in Object) return Boolean;

  function Shaft_Speed (Instance : in Object)
                       return Ct.Radians_Per_Second;

  -- *****************************************************

private

  type Attributes;

  type Object is access Attributes;

end Drive_Unit_Class;
```

drive_unit_class body

```
with Dc_Motor_Class;
with Gear_Box_Class;

with Common_Types;
use Common_Types;

package body Drive_Unit_Class is

  -- ********************* Attributes ********************

  type Motor_Object is new Dc_Motor_Class.Object;
  type Gear_Box_Object is new Gear_Box_Class.Object;

  type Attributes is
    record
      Motor : Motor_Object;
      Gear_Box : Gear_Box_Object;
      Motor_Status : Ct.On_Off := Ct.Off;
    end record;

  Motor_Load : constant Ct.Amps := 1.0;
  Motor_Max_Speed : constant Ct.Radians_Per_Second :=
    628.0;  -- 6000 rpm
  Motor_Max_Torque : constant Ct.Foot_Pounds := 300.0;
  Motor_Max_Volts : constant Ct.Volts := 15.0;
  Motor_Min_Volts : constant Ct.Volts := 8.0;
  Gear_Box_Max_Torque : constant Ct.Foot_Pounds := 700.0;

  -- ********************* Modifiers *********************

  procedure Create (Instance : in out Object;
                    Gearbox_Max_Torque : in Ct.Foot_Pounds) is

  begin
    -- Create the drive unit
    Instance := new Attributes;

    Instance.Motor_Status := Off;

    -- Create the motor
    Create (Instance => Instance.Motor,
            Max_Voltage => Motor_Max_Volts,
            Min_Voltage => Motor_Min_Volts,
            Nominal_Load => Motor_Load,
            Nominal_Speed => Motor_Max_Speed,
            Nominal_Torque => Motor_Max_Torque);

    -- Create the gear box
    Create (Instance => Instance.Gear_Box,
            Max_Torque => Gear_Box_Max_Torque);

  end Create;


  procedure Modify (Instance : in out Object;
                    Command : in Commands) is
  begin
    case Command.Kind is
      when Gearbox_Seizure =>
        declare
          Gearbox_Command : constant Gear_Box_Class.Commands :=
            (Kind => Gear_Box_Class.Gear_Seizure,
             Apply => Command.Apply);
        begin
          Modify (Instance => Instance.Gear_Box,
                  Command => Gearbox_Command);
        end;

      when Motor_Fail =>
        declare
          Motor_Command : constant Dc_Motor_Class.Commands :=
            (Kind => Dc_Motor_Class.Motor_Fail,
             Apply => Command.Apply);
        begin
          Modify (Instance => Instance.Motor,
                  Command => Motor_Command);
        end;
    end case;

  end Modify;


  procedure Periodic_Update (Instance : in out Object;
                             Avail_Power : in Ct.Volts;
                             Delta_Time : in Ct.Seconds;
                             Torque : in Ct.Foot_Pounds) is
  begin
    --  Update electric motor
    Periodic_Update (Instance => Instance.Motor,
                     Delta_Time => Delta_Time,
                     Torque => Torque_Load (Instance.Gear_Box),
                     Avail_Power => Avail_Power);

    -- Set motor Status flag
    if Avail_Power >= 0.1 then
      Instance.Motor_Status := Ct.On;
    else
      Instance.Motor_Status := Ct.Off;
    end if;

    -- Update Gear box
    Periodic_Update (Instance => Instance.Gear_Box,
                     Delta_Time => Delta_Time,
                     Torque => Torque,
                     Supply_Speed => Shaft_Output
                                       (Instance.Motor));

  end Periodic_Update;

  -- ********************* Selectors *********************

  function Elec_Load (Instance : in Object) return Ct.Amps is
  begin
    return (Load (Instance.Motor));
  end Elec_Load;


  function Motor_On (Instance : in Object) return Boolean is
  begin
    return (Instance.Motor_Status = Ct.On);
  end Motor_On;


  function Shaft_Speed (Instance : in Object)
                       return Ct.Radians_Per_Second is
  begin
    return (Shaft_Output (Instance.Gear_Box));
  end Shaft_Speed;

end Drive_Unit_Class;
```

gear_box_class specification

```
with Common_Types;

package Gear_Box_Class is

  type Object is limited private;

  package Ct renames Common_Types;

  -- ******************** Commands *********************

  type Command_Kinds is (Gear_Seizure);

  type Commands (Kind : Command_Kinds := Gear_Seizure) is
    record
      case Kind is
        when Gear_Seizure =>
          Apply : Boolean := False;
      end case;
    end record;

  -- ******************** Modifiers ********************

  procedure Create (Instance : in out Object;
                    Max_Torque : in Ct.Foot_Pounds);

  procedure Modify (Instance : in out Object;
                    Command : in Commands);

  procedure Periodic_Update
              (Instance : in out Object;
               Delta_Time : in Ct.Seconds;
               Torque : in Ct.Foot_Pounds;
               Supply_Speed : in Ct.Radians_Per_Second);

  -- ******************** Selectors ********************

  function Torque_Load (Instance : in Object)
                      return Ct.Foot_Pounds;

  function Shaft_Output (Instance : in Object)
                       return Ct.Radians_Per_Second;

  -- ****************************************************

private

  type Attributes;

  type Object is access Attributes;

end Gear_Box_Class;
```

gear_box_class body

```
package body Gear_Box_Class is

  -- ******************** Attributes ********************

  type Attributes is
    record
      Max_Torque_Load : Ct.Foot_Pounds := 0.0;
      Torque_Load : Ct.Foot_Pounds := 0.0;
      Shaft_Speed : Ct.Radians_Per_Second := 0.0;
      Seized : Boolean := False;
    end record;

  -- ******************** Modifiers **************************

  procedure Create (Instance : in out Object;
                    Max_Torque : in Ct.Foot_Pounds) is
  begin
    Instance := new Attributes'(Max_Torque_Load => Max_Torque,
                                Torque_Load => 0.0,
                                Shaft_Speed => 0.0,
                                Seized => False);
  end Create;


  procedure Modify (Instance : in out Object;
                    Command : in Commands) is
  begin
    case Command.Kind is
      when Gear_Seizure =>
        Instance.Seized := Command.Apply;
    end case;

  end Modify;

  procedure Periodic_Update
              (Instance : in out Object;
               Delta_Time : in Ct.Seconds;
               Torque : in Ct.Foot_Pounds;
               Supply_Speed : in Ct.Radians_Per_Second) is
  begin

    -- Based on shaft status, determine shaft speed and torque load
    if not Instance.Seized then
      Instance.Shaft_Speed := Supply_Speed;
      Instance.Torque_Load := Torque;
    else
      Instance.Shaft_Speed := 0.0;
      Instance.Torque_Load := Instance.Max_Torque_Load;
    end if;

  end Periodic_Update;

  -- ******************** Selectors ********************

  function Torque_Load (Instance : in Object)
                      return Ct.Foot_Pounds is
  begin
    return (Instance.Torque_Load);
  end Torque_Load;


  function Shaft_Output (Instance : in Object)
                       return Ct.Radians_Per_Second is
  begin
    return (Instance.Shaft_Speed);
  end Shaft_Output;

end Gear_Box_Class;
```

hydraulic_pump_class specification

```
with Common_Types;

package Hydraulic_Pump_Class is

  type Object is limited private;

  package Ct renames Common_Types;

  -- ******************** Commands *********************

  type Command_Kinds is (Modify_Flow_Rate, Compensator_Fail,
                         Pump_Fail);

  type Commands (Kind : Command_Kinds := Modify_Flow_Rate) is
    record
      case Kind is
        when Modify_Flow_Rate =>
          Bias : Ct.Non_Dimensional := 0.0;
          Scale : Ct.Non_Dimensional := 1.0;
        when others =>
          Apply : Boolean := False;
      end case;
    end record;

  -- ******************** Modifiers ********************

  procedure Create (Instance : in out Object);

  procedure Modify (Instance : in out Object;
                    Command : in Commands);


  procedure Periodic_Update
              (Instance : in out Object;
               Delta_Time : in Ct.Seconds;
               Fluid_Avail : in Boolean;
               Shaft_Speed : in Ct.Radians_Per_Second;
               System_Pressure : in Ct.Psi);

  -- ******************** Selectors ********************

  function Consumed_Flow (Instance : in Object)
                        return Ct.Gallons_Per_Second;

  function Output_Flow (Instance : in Object)
                      return Ct.Gallons_Per_Second;

  function Pump_On (Instance : in Object) return Boolean;

  function Torque (Instance : in Object) return Ct.Foot_Pounds;
  -- *******************************************************

private

  type Attributes;

  type Object is access Attributes;

end Hydraulic_Pump_Class;
```

hydraulic_pump_class body

```
with Axial_Piston_Pump_Class;
with Actuator_Class;
with Centrifugal_Pump_Class;

with Numeric_Types;
with Common_Types;
use Common_Types;

package body Hydraulic_Pump_Class is

  -- ******************** Attributes *******************

  type Axial_Pump_Object is new Axial_Piston_Pump_Class.Object;
  type Actuator_Object is new Actuator_Class.Object;
  type Centrifugal_Pump_Object is
    new Centrifugal_Pump_Class.Object;

  type Attributes is
    record
      Axial_Pump : Axial_Pump_Object;
      Actuator : Actuator_Object;
      Centrifugal_Pump : Centrifugal_Pump_Object;
      Compensator_Fail : Boolean := False;
      Consumed_Flow : Ct.Gallons_Per_Second := 0.0;
      Flow_Bias : Ct.Gallons_Per_Second := 0.0;
      Flow_Out : Ct.Gallons_Per_Second := 0.0;
      Flow_Scale : Ct.Non_Dimensional := 1.0;
      Press_Delta : Ct.Psi := 0.0;
      Pump_Sensor_Failure : Boolean := False;
      Pump_Status : Ct.On_Off := Ct.Off;
      Shaft_Fail : Boolean := False;
      Shaft_Speed : Ct.Radians_Per_Second := 0.0;
      Torque : Ct.Foot_Pounds := 0.0;
    end record;

  Num_Of_Pistons : constant Natural := 6;
  Single_Piston_Area : constant Ct.Square_Feet :=
    5.45415E-3;   -- 1" diam
  Displacement : constant Ct.Gallons := 0.4922437;

  -- *******************************************************

  -- Overloaded Operators

  function "*" (Left : in Ct.Non_Dimensional;
                Right : in Ct.Gallons_Per_Second)
               return Ct.Gallons_Per_Second is
  begin
    return Ct.Gallons_Per_Second
          (Numeric_Types."*" (Ct.Sp_Real (Left),
                              Ct.Sp_Real (Right)));
  end "*";

  function "*" (Left : in Ct.Psi; Right : in Ct.Non_Dimensional)
               return Ct.Psi is
  begin
    return Ct.Psi (Numeric_Types."*"
                  (Ct.Sp_Real (Left), Ct.Sp_Real (Right)));
  end "*";

  -- ******************** Modifiers ********************

  procedure Create (Instance : in out Object) is

  begin
    -- Create the hydraulic pump
    Instance := new Attributes;

    Instance.Compensator_Fail := False;
    Instance.Consumed_Flow := 0.0;
    Instance.Flow_Bias := 0.0;
    Instance.Flow_Out := 0.0;
    Instance.Flow_Scale := 1.0;
    Instance.Press_Delta := 0.0;
    Instance.Pump_Sensor_Failure := False;
    Instance.Pump_Status := Ct.Off;
    Instance.Shaft_Fail := False;
    Instance.Shaft_Speed := 0.0;
    Instance.Torque := 0.0;

    -- Create the actuator
    Create (Instance => Instance.Actuator);

    -- Create the axial piston pump
    Create (Instance => Instance.all.Axial_Pump,
            Displacement => Displacement,
            Efficiency => 1.0,
            Number_Of_Pistons => Num_Of_Pistons,
            Piston_Area => Single_Piston_Area);

    -- Create the centrifugal pump
    Create (Instance => Instance.Centrifugal_Pump);

  end Create;


  procedure Modify (Instance : in out Object;
                    Command : in Commands) is

  begin
    case Command.Kind is

      when Modify_Flow_Rate =>
        Instance.Flow_Scale := Command.Scale;
        Instance.Flow_Bias := Ct.Gallons_Per_Second (Command.Bias);

      when Compensator_Fail =>
        Instance.Compensator_Fail := Command.Apply;

      when Pump_Fail =>
        Instance.Shaft_Fail := Command.Apply;

    end case;
  end Modify;


  procedure Periodic_Update
              (Instance : in out Object;
               Delta_Time : in Ct.Seconds;
               Fluid_Avail : in Boolean;
               Shaft_Speed : in Ct.Radians_Per_Second;
               System_Pressure : in Ct.Psi) is
  begin

    -- Determine shaft speed based on suppled shaft
    -- speed and shaft status.
    if not Instance.Shaft_Fail then
      Instance.Shaft_Speed := Shaft_Speed;
    else
      Instance.Shaft_Speed := 0.0;
    end if;

    -- Update Centrifugal Pump
    Periodic_Update (Instance => Instance.Centrifugal_Pump,
                     Delta_Time => Delta_Time,
                     Supply_Speed => Instance.Shaft_Speed,
                     Fluid_Avail => Fluid_Avail,
                     Consumed_Flow => Flow (Instance.Axial_Pump));

    -- Calculate pressure difference between scavenge
    -- pump output pressure and system pressure.  Include
    -- effects of pressure compensator failure (if required).
    Instance.Press_Delta := System_Pressure -
                            Pressure (Instance.Centrifugal_Pump);

    if Instance.Press_Delta <= 0.1 then
      Instance.Press_Delta := 0.0;
    end if;

    if Instance.Compensator_Fail then
      -- Cause a strange pressure delta
      Instance.Press_Delta :=
        Instance.Press_Delta *
          Ct.Non_Dimensional (Instance.Flow_Bias);
    end if;

    -- Update pressure compensation actuator
    Periodic_Update (Instance => Instance.Actuator,
                     Delta_Time => Delta_Time,
                     Pressure => Instance.Press_Delta);

    -- Update Axial Piston Pump
    Periodic_Update (Instance => Instance.Axial_Pump,
                     Pressure => Instance.Press_Delta,
                     Stroke => Stroke (Instance.Actuator),
                     Rotation => Shaft_Speed);
    Instance.Flow_Out := Flow (Instance.Axial_Pump);

    Instance.Flow_Out :=
      Instance.Flow_Scale * Instance.Flow_Out + Instance.Flow_Bias;

    -- Update ouput variables
    if Shaft_Speed >= 0.1 then
      Instance.Pump_Status := Ct.On;
    else
      Instance.Pump_Status := Ct.Off;
    end if;

    -- Pump torque includes centrifugal pump and axial piston pump
torque terms
    Instance.Torque := Torque (Instance.Centrifugal_Pump) +
                         Torque (Instance.Axial_Pump);

  end Periodic_Update;

  -- ********************* Selectors ********************

  function Consumed_Flow (Instance : in Object)
                         return Ct.Gallons_Per_Second is
  begin
    return Consumed_Flow (Instance.Centrifugal_Pump);
  end Consumed_Flow;


  function Output_Flow (Instance : in Object)
                       return Ct.Gallons_Per_Second is
  begin
    return (Instance.Flow_Out);
  end Output_Flow;


  function Pump_On (Instance : in Object) return Boolean is
  begin
    return (Instance.Pump_Status = Ct.On);
  end Pump_On;


  function Torque (Instance : in Object) return Ct.Foot_Pounds is
  begin
    return (Instance.Torque);
  end Torque;

end Hydraulic_Pump_Class;
```

numeric_types specification

```
package Numeric_Types is

  type Sp_Real is digits 6;
  type Sp_Integer is range -2_147_483_647 .. 2_147_483_647;

end Numeric_Types;
```

positive_displacement_pump_class specification

```
with Common_Types;

package Positive_Displacement_Pump_Class is

  type Object is limited private;

  package Ct renames Common_Types;

  -- ********************* Commands *********************

  type Command_Kinds is (Set_Efficiency);

  type Commands (Kind : Command_Kinds := Set_Efficiency) is
    record
      case Kind is
        when Set_Efficiency =>
          Efficiency : Ct.Non_Dimensional := 1.0;
      end case;
    end record;

  -- ********************* Modifiers ********************

  procedure Create (Instance : in out Object;
                    Displacement : in Ct.Gallons;
                    Efficiency : in Ct.Non_Dimensional);

  procedure Modify (Instance : in out Object;
                    Command : in Commands);

  procedure Periodic_Update
              (Instance : in out Object;
               Efficiency : in Ct.Non_Dimensional := 1.0);

  -- ********************* Selectors ********************

  function Flow (Instance : in Object)
                return Ct.Gallons_Per_Second;

  function Efficiency (Instance : in Object)
                      return Ct.Non_Dimensional;

  -- *****************************************************

private

  type Attributes;

  type Object is access Attributes;

end Positive_Displacement_Pump_Class;
```

positive_displacement_pump_class body

```
with Numeric_Types;

package body Positive_Displacement_Pump_Class is

  -- ********************* Attributes ********************
  type Attributes is
    record
      Displacement : Ct.Gallons := 0.0;
      Efficiency : Ct.Non_Dimensional := 1.0;
      Flow : Ct.Gallons_Per_Second := 0.0;
    end record;

  -- *****************************************************
  function "*" (Left : in Ct.Gallons;
                Right : in Ct.Non_Dimensional)
               return Ct.Gallons_Per_Second is
  begin
    return Ct.Gallons_Per_Second
              (Numeric_Types."*" (Ct.Sp_Real (Left),
                                  Ct.Sp_Real (Right)));
  end "*";

  -- ********************* Modifiers ********************

  procedure Create (Instance : in out Object;
                    Displacement : in Ct.Gallons;
                    Efficiency : in Ct.Non_Dimensional) is
  begin
    Instance := new Attributes'(Displacement => Displacement,
                                Efficiency => Efficiency,
                                Flow => Displacement * Efficiency);
  end Create;

  procedure Modify (Instance : in out Object;
                    Command : in Commands) is
  begin
    case Command.Kind is
      when Set_Efficiency =>
        Instance.Efficiency := Command.Efficiency;
    end case;
  end Modify;


  procedure Periodic_Update
              (Instance : in out Object;
               Efficiency : in Ct.Non_Dimensional := 1.0) is
  begin
    Instance.Efficiency := Efficiency;
  end Periodic_Update;

  -- ********************* Selectors ********************

  function Flow (Instance : in Object)
                return Ct.Gallons_Per_Second is
  begin
    return Instance.Flow;
  end Flow;

  function Efficiency (Instance : in Object)
                      return Ct.Non_Dimensional is
  begin
    return Instance.Efficiency;
  end Efficiency;
end Positive_Displacement_Pump_Class;
```

pressurization_system_class specification

```
with Common_Types;

package Pressurization_System_Class is

  type Object is limited private;

  package Ct renames Common_Types;

  -- ******************** Commands ********************

  type Command_Kinds is (Modify_Flow_Rate,
                         Gearbox_Seizure, Motor_Fail,
                         Compensator_Fail, Pump_Fail);

  type Commands (Kind : Command_Kinds := Modify_Flow_Rate) is
    record

      Apply : Boolean := False;

      case Kind is
        when Modify_Flow_Rate =>
          Bias : Ct.Non_Dimensional := 0.0;
          Scale : Ct.Non_Dimensional := 1.0;
        when others =>
          null;
      end case;

    end record;

  -- ******************** Modifiers ********************

  procedure Create (Instance : in out Object);

  procedure Modify (Instance : in out Object;
                    Command : in Commands);

  procedure Periodic_Update (Instance : in out Object;
                             Avail_Power : in Ct.Volts;
                             Delta_Time : in Ct.Seconds;
                             Torque : in Ct.Foot_Pounds;
                             Fluid_Avail : in Boolean;
                             System_Pressure : in Ct.Psi);

  -- ******************** Selectors ********************

  function Consumed_Flow (Instance : in Object)
                         return Ct.Gallons_Per_Second;

  function Output_Flow (Instance : in Object)
                       return Ct.Gallons_Per_Second;

  function Pump_On (Instance : in Object) return Boolean;

  function Elec_Load (Instance : in Object) return Ct.Amps;

  function Motor_On (Instance : in Object) return Boolean;

  -- *****************************************************

private

  type Attributes;

  type Object is access Attributes;

end Pressurization_System_Class;
```

pressurization_system_class body

```
with Drive_Unit_Class;
with Hydraulic_Pump_Class;

package body Pressurization_System_Class is

  -- ******************** Attributes ********************

  type Drive_Unit_Object is new Drive_Unit_Class.Object;
  type Hydraulic_Pump_Object is new Hydraulic_Pump_Class.Object;

  type Attributes is
    record
      Drive_Unit : Drive_Unit_Object;
      Hydraulic_Pump : Hydraulic_Pump_Object;
    end record;

  Max_Torque : constant Ct.Foot_Pounds := 675.0;

  -- ******************** Modifiers ********************

  procedure Create (Instance : in out Object) is
  begin

    Instance := new Attributes;

    Create (Instance => Instance.Drive_Unit,
            Gearbox_Max_Torque => Max_Torque);

    Create (Instance => Instance.Hydraulic_Pump);

  end Create;


  procedure Modify (Instance : in out Object;
                    Command : in Commands) is
  begin
    case Command.Kind is
      when Modify_Flow_Rate =>
        declare
          Temp_Command : constant Hydraulic_Pump_Class.Commands :=
            (Kind => Hydraulic_Pump_Class.Modify_Flow_Rate,
             Bias => Command.Bias,
             Scale => Command.Scale);
        begin
          Modify (Instance.Hydraulic_Pump, Temp_Command);
        end;

      when Gearbox_Seizure =>
        declare
          Temp_Command : constant Drive_Unit_Class.Commands :=
            (Kind => Drive_Unit_Class.Gearbox_Seizure,
             Apply => Command.Apply);
        begin
          Modify (Instance.Drive_Unit, Temp_Command);
        end;

      when Motor_Fail =>
        declare
          Temp_Command : constant Drive_Unit_Class.Commands :=
            (Kind => Drive_Unit_Class.Motor_Fail,
             Apply => Command.Apply);
        begin
          Modify (Instance.Drive_Unit, Temp_Command);
        end;

      when Compensator_Fail =>
        declare
          Temp_Command : constant Hydraulic_Pump_Class.Commands :=
            (Kind => Hydraulic_Pump_Class.Compensator_Fail,
             Apply => Command.Apply);
        begin
          Modify (Instance.Hydraulic_Pump, Temp_Command);
        end;

      when Pump_Fail =>
        declare
          Temp_Command : constant Hydraulic_Pump_Class.Commands :=
            (Kind => Hydraulic_Pump_Class.Pump_Fail,
             Apply => Command.Apply);
        begin
          Modify (Instance.Hydraulic_Pump, Temp_Command);
        end;

    end case;
  end Modify;


  procedure Periodic_Update (Instance : in out Object;
                             Avail_Power : in Ct.Volts;
                             Delta_Time : in Ct.Seconds;
                             Torque : in Ct.Foot_Pounds;
                             Fluid_Avail : in Boolean;
                             System_Pressure : in Ct.Psi) is


  begin

    -- Update the drive unit
    Periodic_Update (Instance => Instance.Drive_Unit,
                     Avail_Power => Avail_Power,
                     Delta_Time => Delta_Time,
                     Torque => Torque);

    -- Update the hydraulic pump
    Periodic_Update (Instance => Instance.Hydraulic_Pump,
                     Delta_Time => Delta_Time,
                     Fluid_Avail => Fluid_Avail,
                     Shaft_Speed => 107.23,
                     System_Pressure => System_Pressure);

  end Periodic_Update;

  -- ********************* Selectors *********************

  function Consumed_Flow (Instance : in Object)
                         return Ct.Gallons_Per_Second is
  begin
    return Consumed_Flow (Instance.Hydraulic_Pump);
  end Consumed_Flow;

  function Output_Flow (Instance : in Object)
                       return Ct.Gallons_Per_Second is
  begin
    return Output_Flow (Instance.Hydraulic_Pump);
  end Output_Flow;

  function Pump_On (Instance : in Object) return Boolean is
  begin
    return Pump_On (Instance.Hydraulic_Pump);
  end Pump_On;

  function Elec_Load (Instance : in Object) return Ct.Amps is
  begin
    return Elec_Load (Instance.Drive_Unit);
  end Elec_Load;

  function Motor_On (Instance : in Object) return Boolean is
  begin
    return Motor_On (Instance.Drive_Unit);
  end Motor_On;

end Pressurization_System_Class;
```

References:

[1] Shyam R. Chidamber and Chris F. Kemerer, "Towards a Metrics Suite for Object Oriented Design," Proceedings of OOPSLA '91, in ACM SIGPLAN Notices, pp. 197-211, 1991.

[2] Thomas McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. SE #2, pp. 308-320, December 1976.

[3] Joseph Policella, Joey White and Keith Shillington, "An Object-Oriented Command and Telemetry Black Box Simulation Using Ada," Proceedings of the International Telemetering Conference, pp. 1-6, 1993.

About the authors:

Bill Fay is a software engineer employed with CAE Link Space Operations in Houston, Texas and is currently working on the Space Station Verification and Training Facility (SSVTF) project at the Johnson Space Center. Mr. Fay holds a BS Electrical Engineering from Michigan State University and over the past six years has worked on various commercial and military avionics and defense systems implemented in Ada

Viktor Ohnjec is a software engineering consultant employed with Rational and is involved with several large-scale Ada projects including the Space Station Verification and Training Facility (SSVTF) and the Canadian Automated Air Traffic Control System (CAATS). Mr. Ohnjec received his BE Computer Engineering from Concordia University.

A DATA FROM ADA LABORATORY

Key words: Ada, Software Metrics, Software Measures, Software Science, Language Level, Program Difficulty.

Author: Aftab Ahmad.

Address: Dr. Aftab Ahmad,
Department of Computer Science,
College of Computer and Information Sciences,
King Saud University, P.O.Box-51178, Riyadh-11543,
Saudi Arabia.

Telephone: (Office): 966-1-4676581
(Home): 966-1-4682467

Fax Number: 966-1-4675423

Email Address: F60C015@SAKSU00.BITNET

ABSTRACT

Ever since the advent of Ada, it is strongly believed that Ada is not just another programming language. A great deal of literature indicates Ada's uniqueness in having distinctive technical and non-technical characteristics among current and past high-level programming languages. However, in reality little empirical evidence is available to support these assertions. It appears logical to conduct a quantitative comparative experimentation and evaluation of Ada's merits versus that of other high-level programming languages. In this paper, we have presented the results of our long range quantitative experimentation involving data-collection and application of software science measurements and metrics on Ada programs, collected from academic environments and published literature. As a part of this effort, the author has developed an intelligent system referred to as " Automated Ada Laboratory (AAL)", which integrates many tools dealing with data-collection and computation of software measurements and metrics from Ada program samples. This paper reports the results of application of important software science metrics. We have tried to assess the level of Ada language with respect to previously published language levels. Our study confirms quantitatively, Ada's strength relative to other programming languages with respect to important program characteristics of language level and program difficulty.

1: Introduction

The magnitude of growing costs invested towards management of Software life cycle is making it essentially important to improve software product quality. The process of characterization and comparing factors affecting programming effort is not possible without application of measurements and metrics. During the past decades a comparable effort has been devoted to the area of software engineering for the enhancement of software development process by refinement of software design and implementation styles. On the other hand, an equally important effort has been devoted in the field of programming languages for the design and implementation of new programming languages.

The role of analysis and design phases is important for development of economical software product. However programming language available for implementation will have serious impact on the quality of the end product . The cost involving software implementation, testing and maintenance mainly depends on the programming language used.

Shneiderman [1] has proposed a syntactic-semantic model of programming process indicating the significance of programming languages on coding and maintenance phases of a given software system.

It is strongly believed that Ada is not just another programming language[2,3] It has distinctive technical and non-technical characteristics among current and past high-level languages. Ada was specifically developed in response to a crisis in software development. Ada embodies many modern software engineering methods, and is expected to be the language with remarkable power of expression. Various experimental studies have indicated that Ada enhances the software development process and provides abstraction better than many programming languages. However, it is important to indicate that without quantitative measurements, deciding whether Ada has provided any improvement over the current and past programming languages may be a highly subjective matter. This paper presents an interesting and important empirical study involving data-collection and application of software measurements and models on Ada programs. The area of software metrics and models [4,5,6] nicely provides basis for measurements and experimentation in this direction. Within a large body of available [6] software metrics; software science metrics [4,5] has been accepted intellectually appealing. Software science metrics provide rigorous basis for characterization and comparison of programming languages' characteristics relating to programming process. It proposes hypothesis to relate a particular language facility to the mental effort required to use it. It provides metrics to objectively estimate the power of a given programming language and its influence on the human effort for implementation and maintenance of a program.

The theory of software science [4] applies scientific methods to the software development and analysis . Various experimental studies provide evidence [4] to support the theory. A computer program is considered in software science to be a series of tokens, which can be classified as operators and operands. All software science metrics depend on four primitive measures namely: ***n1 (Number of distinct operators), n2 (Number of distinct operands), N1 (Total number of operators), N2 (Total number of operands). The size of the vocabulary n is defined to be n1+n2 and the length of the program N to be N1+N2.***

Software science proposes a number of program impurities which reduces the effectiveness of program by increasing program length unnecessarily. The metric ***"Estimated Length" EN*** is an estimate of length of pure program and is given by:

$$EN= n1*Log_2\ n1 + n2*Log_2\ n2$$

The ***program volume V*** measures the number of bits required to specify a program in a given programming language. The volume V is defined as:

$$V=N*Log_2\ n$$

Software science provides ***"Program Level" (L)*** metric to asses level of abstraction in a given program. Software Science provides estimated program level ***(EL)*** given by:

$$EL = (2/n1)*(n2/N2)$$

The ***"Program Difficulty"*** metric ***D2*** is defined as the estimate of human difficulty in the implementation of an algorithm in a given programming language. It can be simply calculated by ***1/EL***. The software science metric ***"Effort" (E)*** attempts to quantify the effort required to comprehend the existing implementation of an algorithm. This metric represents the number of mental discriminations (decisions) that a fluent, concentrating programmer should make to comprehend the implementation of an algorithm in a given language. The metric ***E*** can be computed by ***V/EL***. The ***potential volume (or Intelligence Content) IC*** is a metric that measures inherent content of a program and is given by ***EL*V***. Another important aspect of software quality which has been proposed and identified by software science is ***"Language Level" (λ)*** to measure the inherent limitation imposed by the language . The value of metric λ is given by $EL^2 *V$. A high value of language level reflects its power to decrease human difficulty for the implementation of a given algorithm. It predicts the level of abstraction provided by a programming language. It also provides basis to rank languages on a linear scale. A series of studies have been performed [4-14] to predict the language levels of various programming languages. These levels are summarized as under [4]:

Language	*Language Level*	*Standard Deviation*
English	*2.16*	*0.86*
APL	*1.98*	*1.36*
PL/1	*1.53*	*0.96*
Algol 58	*1.21*	*0.86*
Fortran	*1.41*	*0.9*
Pilot	*0.92*	*0.66*
Assembler	*0.88*	*0.65*
IBM/370 BAL	*0.43*	*0.83*

TABLE: 1

It would be interesting to investigate Ada's position in this table. Intuitively, Ada is thought to be of a higher level than various number of existing programming languages. Therefore, experimentation for evaluation of Ada's level is important for producing evidence to support or contradict software science.

The analysis of results from our experimentation of software science metrics including Program Difficulty is also reported in this paper. The distribution of operators and operands, which plays important role towards program comprehension and maintenance is exposed for Ada language from our experimentation.

2: Automated Ada Laboratory (AAL):

The author has developed an intelligent System referred to as "Automated Ada Laboratory (AAL), which integrates many tools dealing with data-collection and computation of software science measurements and metrics from ada program samples. AAL is capable of performing variety of analysis tasks including storing and retrieving important bodies of data , static analysis, dynamic analysis, software quality assurance, metric analysis, and control-flow structural analysis. The overall environments provide an excellent opportunity for ongoing research and experimentation towards quantitative evaluation and finding possible relationships among measurements for ada programming language.

To simplify the use of the system, a window-based user- friendly interface gives user nice assistance and provides capability of answering questions and performing desired analysis tasks. The use of tools is reasonably simple and their representations are not visible to the user. AAL provides all database manipulation facilities.

The database is the most important component of AAL since it serves as the unifying elements for all other components of AAL. The database holds various analysis tools and libraries of ADA program samples user.

The Ada program samples libraries are maintained in a classified manner. Each program sample is indexed according to size, type of environments where is produced and its application area. This allows analysis of all available program samples depending on any chosen index. For instance, all program samples dealing with data-structures application can be analyzed.

The major purpose of AAL is to establish feedback loop by providing a broad and diverse base of analysis and measurements on Ada programs. Several quantitative answers are readily available covering wide spectrum of Ada characteristics research towards quantitative evaluation and finding possible relationships among measurements for Ada programming language. The Ada laboratory has been used to analyze the software science metrics and various standard program measures.

3: Analysis of Data:

The application of software science metrics[4] on algorithms is straight forward as each symbol can be strictly classified as operator or operand. The process of classification may have some problems when algorithm is represented in computer programming languages. Some previous studies [4,9] have indicated effects of counting methods of operators and operands on software science measurements. However, there is a general agreement now [4] that as long as consistency is maintained, small details do not matter.

For Ada language, AAL maintains the dynamic process of operator and operand classification for each symbol in Ada program. User of AAL has facility to choose any desired classification for all symbols in Ada program prior to invoking analysis operation.

For the purpose of this experiment, the classification scheme given in Table: 2 has been utilized for the results reported in this paper.

Ada Lexeme	*Classification*
All special symbols	*Operators*
Reserved word Identifiers	*Operators*
Program Identifiers	*Operators*
Attribute Identifiers	*Operators*
Type Identifiers	*Operands*
Variable Identifiers	*Operands*
Constant Identifiers	*Operands*
Subroutine Identifiers	*Operands*
Exception Identifiers	*Operands*
Task Entry Identifiers	*Operands*
Task Identifiers	*Operators*
Package Identifiers	*Operators*
Numeric Literals	*Operands*
String Literals	*Operands*
Character Literals	*Operands*

Table: 2

Large number of Ada program samples have been collected from academic environments and published literature over the period of past few years. The available program samples cover wide area of application including: Embedded Systems, and Abstract Data Type operations.

An undergraduate course entitled "Special Topics" focusing on Ada programming and its applications offered by the Computer Engineering. Department at King Saud University was our main source of program sample collection. Other program samples were obtained from student projects and faculty staff.

AAL was used to apply software science metrics including n1, n2, N1, N2, n, N, EN, V, EL,E, IC, and λ on all Ada program samples available currently in our databases. For the purpose of demonstration, data on these Software Science is given for twenty Ada packages in Table:3. The analysis is available on the software science metric λ will be discussed with particular emphasis.

The statistical analysis of software science metrics obtained from evaluation of all available ada program samples present interesting results. The distribution of operators and operands have significant influence on various important program characteristics including program difficulty. The averages of n1, n2, N1, and N2 came out 25.15, 33.1, 187.8 and 87.6 respectively. Software science theory[4] predicts that the lower values of n2 and N2 strongly influence the reduction of program difficulty and program bugs. Since average of distinct operators tend to be stable in our experimentation. Therefore the ratio N2/n2 appears to be main contributor to the program difficulty metric D2. The ratio N2/n2 from our experimentation is 2.646. This value is lower than the corresponding values of ratio N2/n2 published by previous studies analyzing other programming languages. The distribution of operators and operands in Ada language appears to enhance program implementation by reducing program difficulty.

The average of λ from overall experimentation came out 1.69 with standard deviation of 0.73. This value nicely places Ada on the top of Table:1, with the exception of English language having higher value of 2.16. Furthermore, It is observed (also indicated in Table:3), that the λ does not have a tendency to increase with program length N. Therefore it further strengthens the fact that Ada has higher language level than other programming languages shown in Table:1.

The correlation coefficient between N and EN is higher value of 0.971. It nicely verifies software science length equation. The averages of volumes measured for Ada language appears APL [ ] study and lower than PL/1 [ ] study.

Summary

Our experimentation has strongly validated Software Science theory and confirmed quantitatively that Ada has higher level of abstraction than various existing programming languages. We are in process of further data-collection and analysis on Ada programs for the further requirements of the results. The development of AAL provides base for continuos experimentation on Ada's measurable characteristics.

The work proposed can be extended in various directions. The application of control flow structural metrics would be interesting. The results of Software Science Metrics of effort may be verified through psychological experimentation.

REFERENCES

[1]: B.Shneiderman, Software Psychology, Winthrop Publishers, 1980.
[2]: E.Sammet, "Why Ada is not just another Programming Language", CACM, Vol. 29, No. 8, August, 1986.
[3]: J.D.Gannon, E.E.Kartz and V.R.Bassili, "Metrics For Ada Packages: An Initial Study", CACM, Vol. 29, No. 7, July 1986.
[4]: Perlis, Sawayed,and Shaw, Software Metrics: An Analysis and Evaluation, The MIT Press, 1981.
[5]: V.Y.Shen, S.D.Conte and H.E.Dunsmore, "Software Science Revisited: A Critical Analysis of the Theory and its Empirical Support", IEEE-TSE, Vol. SE-9, No.2, March 1983.
[6]: M.H.Halstead, Elements of Software Science Library, Elsevier North Holland Inc., 1977.
[7]: M.H.Halstead, "Natural Laws Controlling Algorithm Structure." ACM SIGPLAN Notices, 7, No.2 Feb 1972, 19-26.
[8]: M.H. Halstead, "Software Physics Comparison of a Sample Program in DSL ALPHA and COBOL". IBM Research Report, RJ 1460 Oct 1974.
[9]: M.R.Woodward, "The Application of Halstead Software Science Theory to ALGOL 68 Programs." Software-Practices and Experiences. 14, No.3 , March 1984.
[10]:De Kerf, "APL and Halstead's Theory of Software Metrics." in APL 81 Conf. Proc. Oct. 1981, 89-93.
[11]: A.H.Konstan and D.E.Wood "Software Science Applied to APL." IEEE Software Engineering, SE-11 No.10 Oct 1985, 994-1000.
[12]: S.H.Zweben, and K.C.Fung "Exploring Software Science Relation in Cobol and APL." Proc.IEEE COMPASAC '79 Nov.1979,702-709.
[13]: S.D.Conte, "The Software Science Language Level Metric." Dept. Comput. Sci., Prude Univ., Rep. CSDTR-373 1981.
[14]: S.H.Zweben, and M.H.Halstead, "The Frequency Distribution of Operators in PL/1 Programs " IEEE Trans.on Soft. Engg., SE-5, No.2 , March 1979, 91-95.

Sofware Science Metrics Values For Twenty Ada Packages

Pkg #	n1	n2	N1	N2	n	N	EN	V	EL	IC	E	λ
1	33	59	300	115	92	415	513.54	2,707.27	0.03	84.17	87,068.82	2.62
2	20	25	91	52	45	143	202.53	785.33	0.05	37.75	16,334.96	1.82
3	23	30	184	100	53	284	251.24	1,626.72	0.03	42.43	62,357.96	1.11
4	26	51	417	268	77	685	411.5	4,292.74	0.01	62.83	293,253.66	0.92
5	22	9	58	22	31	80	126.63	396.33	0.04	14.73	10,657.02	0.55
6	14	9	44	19	23	63	81.83	284.98	0.07	19.28	4,211.43	1.31
7	19	9	51	14	28	65	109.23	312.47	0.07	21.14	4,617.73	1.43
8	12	6	29	7	18	36	58.52	150.11	0.14	21.44	1,050.82	3.06
9	37	30	128	49	67	177	339.95	1,073.69	0.03	35.53	32,443.56	1.18
10	14	7	36	16	21	52	72.95	228.4	0.06	14.27	3,654.4	0.89
11	24	25	132	44	49	176	226.13	988.18	0.05	46.78	20,870.55	2.22
12	48	85	647	330	133	977	812.87	6,893.01	0.01	73.97	642,266.43	0.79
13	19	13	55	28	32	83	128.81	415	0.05	20.28	8,491.53	0.99
14	22	14	75	24	36	99	151.41	511.82	0.05	27.14	9,651.51	1.44
15	37	16	111	42	53	153	256.74	876.37	0.02	18.04	42,558.8	0.37
16	32	60	272	77	92	349	514.41	2,276.72	0.05	110.87	46,748.71	5.4
17	10	4	22	4	14	26	41.21	98.99	0.2	19.71	494.95	3.96
18	13	9	39	24	22	63	76.63	280.94	0.06	16.2	4,869.69	0.94
19	11	12	60	40	23	100	81.07	452.35	0.05	24.67	8,293.19	1.35
20	67	189	1,005	477	256	1,482	1,835.6	11,856	0.01	140.22	1,002,396.57	1.66

TABLE: 2

ADA INTEGRATION FOR AN MS-WINDOWS TOOL FRAMEWORK

Paul R. Pukite
DAINA
4111 Central Ave. NE, Suite 212
Columbia Heights, MN, 55421
pukite@vz.cis.umn.edu

Summary

With the power inherent in many of the current computing environments, including Microsoft (MS) Windows, developers will demand alternative languages specifically designed to handle large and complex projects. Clearly, Ada manages large projects well, but its usefulness for Windows applications remains undocumented. This paper describes techniques used in developing an integrated set of interactive tools for system design. The Ada language and a graphical tool-kit served as a foundation during the development effort. The key result was that large, well-integrated, reliable, maintainable, and reusable Ada applications can be developed for the Windows environment. The development process becomes more manageable and less obscure by using standard Ada constructs such as package specifications, generics, tasking, and exceptions. Furthermore, the use of automatic code generation techniques provided advantages for many of the mundane interface coding tasks.

Objective

This work was part of a development project to provide integrated reliability and performance modeling tools for system designers (i.e. a tool framework). The key design targets were:

1. A graphical and interactive personal computer environment (i.e. MS-Windows).
2. Port-oriented dynamic linking and data exchange between tools.
3. Multiple language application interfaces.
4. Expert system tool command and control.

This paper will describe how the first three targets were met. As an added benefit, the expert system provided a good methodology for automatic code generation for the graphical environment.

Approach

Ada, targeted for Windows applications, was chosen as a development language for our tools. This has proven to be a good match to our application domain as Ada includes robust features such as packaging, generics, information hiding, constraint checking, exception handling, and strong typing. These are all needed for reliable and maintainable code. However, if language integration or interoperability is desired, the Windows environment still allows a standard method – dynamic link libraries (DLL) – for parameter passing and calling conventions. For example, we have tested Ada ↔ C++ interoperability through DLLs defined for several tools, and find that it works effectively.

Our tool communication framework relies on standard Windows features such as dynamic data exchange (DDE). This requires each tool to specify a common message format. Ports can then be defined which map to the format and allow other applications (within the same computer or possibly connected on a network) to access data and/or control the tool. This is an object-based approach where components are plugged into a virtual backplane.

In the next sections, the techniques used will be described in the context of two specific Ada Windows tools (a code timing and tracing tool and a specification parser). The tools are geared to the software development process, are fairly small in size, and are available in the public domain as reusable Windows components.

Windows Environment and Ada

Developing for message-passing, event-driven operating systems such as MS-Windows requires a different approach from that used on text-based input/output applications. This is mainly because of the need to merge a procedurally-oriented language, like Ada, with the Windows system runtime. However, once the developer learns how the runtime interacts with the main program and how the associated library provides the graphical user interface (GUI) services, the remaining application-specific code can be developed according to established conventions.

As an example, when a user interacts with a Windows application program by entering a keyboard or mouse operation, the runtime transmits a message to the application. This message must be handled according to the Windows interface conventions. Thus, the handling of message passing must be an integral part of *any* Windows application. In this respect, Windows parallels the Ada view of control; whereby the Windows runtime serves a similar purpose as the Ada tasking runtime (which is responsible for task scheduling, etc.).

To further the analogy, consider that a non-Ada main program cannot, in general, communicate with an Ada subprogram. This, is due to features of the tasking runtime, elaboration, and exception handling[1]. Likewise, the same conceptual limitation occurs for a non-Windows-compatible language and the Windows interface, where *now* Windows becomes the central dispatcher. To make the language compatible, special hooks must be provided to allow it to receive Windows messages. These hooks or "callback" functions serve as entry ports to the application program (analogous to Ada entry statements).

Aside from using compiler pragmas to deal with these special constructs, the overall program design complexity can be reduced through proper specification, modularity, and reuse. In this regard, Ada provides arguably better facilities for handling large Windows programs than do the current popular languages for Windows development (C/C++).

In particular, two methodologies illustrate how modularity can provide several benefits within the Windows environment. The first deals with code generation and generics when interfacing with Windows dialog boxes. The second presents an inter-application data exchange mechanism featuring DDE, DLL, and Ada tasking.

Dialog Box Interface

A Windows dialog box is the ubiquitous layout form in the Windows environment. Dialog boxes typically serve to select options for an application function.

The developer creates the boxes and corresponding interface code. Tools for creating the boxes and their associated controls (buttons, text, etc.) make the layout task easier, but few tools exist for interfacing to Ada code. One practical scheme relies on Ada specifications[2]. In this method, package specifications provide a foundation for dialog boxes: whereby one considers applying a one-to-one mapping between Ada types and dialog controls. Automatic code generation and the use of generics can then eliminate the remaining manual interface coding.

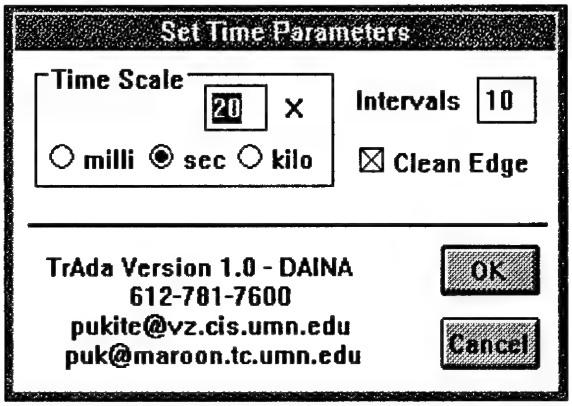
To briefly outline the method, a procedure specification contains a set of arguments corresponding to each of the dialog box controls (check box, radio button, edit text, etc.) and the desired direction of data transfer. The utility of this approach becomes apparent when we use a code generator to create the body, which contains the interface code and the required callback function. Ada's strong typing and attribute mechanisms play a key role in allowing code to be generated with a minimum of specification.

Since the interface specification is used by the Ada compiler, this method is not a preprocessing step as typified by a Classic-Ada preprocessor[3]. However, the same benefits apply. For example, the user is separated from the core functions of `pragma` directives and system details. Further, the style encourages rapid prototyping and reduces GUI programming effort. Lastly, porting the specification to other windowing systems only requires alternate internal rules for the expert-system-based code generator.

Example - Dialog Box Interface Code Generator: Consider, for example, interactively setting parameters for a graph display as in Figure 1 (this is a dialog box from a code tracing and timing analysis tool called *TrAda*). Three options are available:

1. Set the time scale.
2. Set the number of intervals on the time scale.
3. Clean Edges places "nice" values on the time

scale endpoints.



Set Time Parameters

Time Scale 20 ×

○ milli ● sec ○ kilo

Intervals 10

☒ Clean Edge

TrAda Version 1.0 - DAINA
612-781-7600
pukite@vz.cis.umn.edu
puk@maroon.tc.umn.edu

Figure 1: Dialog box for setting graphing options

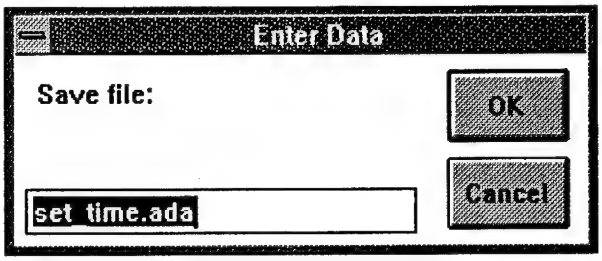
The box was created through a standard visually-oriented Windows resource construction tool. However, the standard tool does not provide the Ada glue or interface code: this is where the specification-to-body code generator plays a role.

Figure 2 shows the Ada specification for the dialog box in Figure 1. First, note the mapping of integer, enumeration type, and boolean arguments to the corresponding dialog box controls. Second, note that the Ada body (not shown), can be automatically generated to provide the glue code for the previously constructed dialog box. Further details are described elsewhere[4].

```
with Wintypes;
package Set_Time is

  type Scale is ( Milli, Sec, Kilo );

  procedure Box
           ( Window     : in     Wintypes.HWND;
             Max_Time   : in out INTEGER;
             Multiplier : in out Scale;
             Intervals  : in out INTEGER;
             Clean_Edge : in out BOOLEAN );

end Set_Time;
```

Figure 2: Ada Specification of dialog box

Example - Generic Dialog Boxes: For dialog boxes that are used frequently, a generic specification can replace the automatically generated code. Figure 3 shows a dialog box that asks for a string input. Figure 4 is a generic package specification which features an abstract data type for the string. An instantiation of the generic would require matching the generic formal objects to the dialog box resource name (`Name`) and dialog box control identifiers (`Prompt_ID` and `Input_ID`).



Enter Data

Save file:

set time.ada

Figure 3: A generic data input dialog box

```
with Wintypes; use Wintypes;
generic
   Name      : in STRING;
   Prompt_ID : in INTEGER;
   Input_ID  : in INTEGER;
   Max_Input : in INTEGER := 128;
package Data_Handler is
   subtype Index is INTEGER range 0 .. Max_Input;
   type Edit_String is private;
   procedure Set(Object : in out Edit_String;
                 Value  : in     STRING);
   function Value(Object : Edit_String)
                   return STRING;
   function Length(Object : Edit_String)
                    return Index;
   procedure Get_String
          (Window    : in     HWND;
           Str       : in out Edit_String;
           Prompt    : in     STRING;
           Highlight : in     BOOLEAN := TRUE);
   procedure Get_Integer
          (Window    : in     HWND;
           Number    : in out INTEGER;
           Low, High : in     INTEGER;
           Prompt    : in     STRING  := "");
private
   subtype Max_String is STRING(1..Max_Input);
   type Edit_String is
      record
         Val : Max_String;
         Len : Index;
      end record;
end Data_Handler;
```

Figure 4: A generic specification for dialog box

Overall, this method more closely follows the conventional way of creating reusable objects through Ada language constructs, but does not allow as much flexibility as a code generator.

Interprocess Communication

Typically, each unique operating system provides its own specific method for inter-program data communication. In the Windows environment, applications usually support the dynamic data exchange (DDE) implementation of interprocess communication. Although the protocol is somewhat complex, DDE uses the standard Windows message capabilities, and thus has long-term potential for tool integration.

As an example of data exchange in action, consider a client-server tool. In a typical case, a consumer wishes to receive data in the form of tokens from a producer application. This is an on-demand service, whereby the producer does not use processing power unless needed.

This type of application conforms nicely to Ada tasking and Windows DDE. Tasking enhances the logical flow of the program, eliminating the need for Windows constructions such as busy waiting on message loops or the use of timer library functions.

Briefly, in the tool framework approach, generic client and server package bodies encapsulate the details of the DDE protocol. The internal processing involves setting up ports and channels for the communication and either posting request messages or waiting for data or acknowledgments. The technique demonstrates how encapsulation, generics, and tasking can hide much of the Windows-specific details. Details of the Ada implementation can be found elsewhere[4].

DLL Specification: By encapsulating the DDE protocol in a wrapper, we can hide even more of the details of the DDE code. This is most easily accomplished by creating a compact dynamic link library (DLL) interface to the DDE ports for a specific application. Providing a DLL wrapper around the DDE ports has several benefits:

1. Linked code library (i.e. import library) remains small.
2. Any DLL-compliant languages can call the library.
3. The tool remains executable as a stand-alone program and thus one can use conventional development methods.

Unfortunately, some restrictions apply to the use of DLLs for Ada programming. These are mostly related to the tasking, elaboration, and exception handling issues mentioned earlier and which affect language interoperability. However, for applications which require a concise DDE wrapper, as proposed here, these constraints are not severe.

Example - *TrAda* DLL: *TrAda* is a tracing and timing analysis DLL for use during Windows program development. In a typical situation, one can use *TrAda* as a debugging tool for monitoring Ada tasking or sequential code flow. *TrAda* works by supplying a DDE message link via a DLL to a client application. By inserting a monitoring symbol string to the Ada code, one can follow the timing behavior of the executing program. In other words, *TrAda* acts as a Windowed oscilloscope trace of program execution.

During execution, *TrAda* acts as the server and the targeted Ada application is the client. Figure 5 shows an Ada binding[5] to the DLL.

```
-- Meridian OpenAda binding for TrAda
with Wintypes;
package TrAda is

  procedure Trace_Name(Name : in STRING); --$STR
  pragma INTERFACE(WINDOWS, Trace_Name,
                           "TrAda_Trace_Name" );

  function Link(Window : in Wintypes.HWND)
       return BOOLEAN;
  pragma INTERFACE(WINDOWS, Link, "TrAda_Link");

end TrAda;

package body TrAda is
end TrAda;
```

Figure 5: DLL specification for tracing tool

The DLL interface is defined through an import library via the `pragma INTERFACE(WINDOWS,...` directives. By replacing the DLL defined names (which occupy a global name space) with the package specification names, we achieve a better encapsulation of the available server subprograms.

In addition, note that the comment line indicated by a `--$STR`, should actually be read as a compiler directive to indicate an Ada `STRING` is being converted to a C-style pointer with a NUL termination. A more portable way of doing this would involve supplying the string conversion in the package body. In general, standardization of compiler directives and pragmas should be an important area for Windows Ada compiler vendors to agree on.

To use *TrAda* the following steps are invoked:

1. The procedure `TrAda.Link(Window)` initializes the link client to *TrAda* and starts the program if it is not already loaded. `Window` is the client window's handle.
2. The `TrAda.Trace_Name("Name")` procedure sends the monitoring symbol via DDE to the concurrently executing *TrAda* program.

```
-- Example test code
with TrAda;
...
function Test_Window_Call ( Window   : HWND;
                            Message  : UINT;
                            wParam   : UINT;
                            lParam   : LONG )
                 return LONG is
begin
   case Message is
      -- window creation
      when Windows.WM_CREATE =>
         if TrAda.Link ( Window ) then
            Con_IO.Put_Line("Linked OK");
         end if;
      -- command from application menu
      when Windows.WM_COMMAND =>
         case wParam is
            when 101 =>
               TrAda.Trace_Name ( "test 101" );
            when 102 =>
               TrAda.Trace_Name ( "test 102" );
            ...
```

Figure 6: Test code for tracing flow

Figure 6 is an example of test code to be traced. When menu commands with identifiers of 101 and 102 are selected, corresponding symbols are sent to *TrAda* via the DDE connection. Figure 7 is a screenshot of the *TrAda* executable with a typical output trace. The `Reset` button resets the time and trace. The `Time` button brings up the *Set Time Parameters* dialog box (see Figure 1).

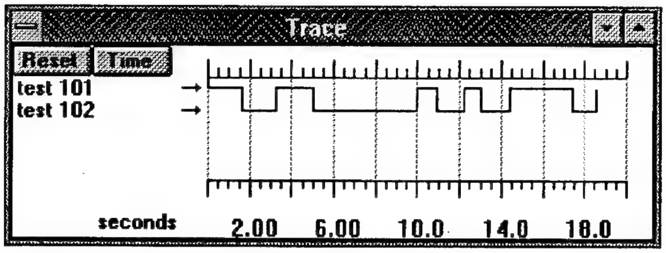


Figure 7: Example trace output

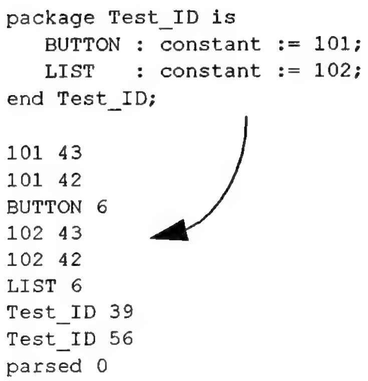
The strength of the DLL approach is that the same library can be used with other DLL-compliant languages, leading to a uniform tool framework.

<u>Example - *SpecPars* DLL:</u> The last example is of a DLL for a Ada specification parser – *SpecPars*. The tool operates by loading an Ada package specification and then providing the client application with a stream of tokens – one at a time. The Ada is parsed according to a grammar modified to handle Ada specifications exclusively. The implementation of *SpecPars* uses a tasking `accept` statement to provide the client with a token name and Ada identifier on demand.

```
with Wintypes;
package SpecPars is
  function Link(Window : in Wintypes.HWND)
                return BOOLEAN;
  pragma INTERFACE(WINDOWS, Link,
                      "SpecPars_Link");
  procedure Load_File(Name : in STRING); --$STR
  pragma INTERFACE(WINDOWS, Load_File,
                      "SpecPars_Load_File");
  procedure Get_Token
           (Token_Name : out STRING; --$STR
            Identifier : out INTEGER);
  pragma INTERFACE(WINDOWS, Get_Token,
                      "SpecPars_Get_Token" );
  END_ID         : constant := 0;
  PRAG           : constant := 1;
  OBJECT         : constant := 2;
  OBJECTC        : constant := 3;
  ARRAYOBJECT    : constant := 4;
  ARRAYOBJECTC   : constant := 5;
  OBJSTATIC      : constant := 6;
  FULLTYPE       : constant := 7;
  ENUMER         : constant := 8;
  ...
end SpecPars;

package body SpecPars is
end SpecPars;
```

Figure 8: Specification parser DLL binding

Figure 8 is the DLL for *SpecPars*. Figure 9 is the output parse for an example specification, showing the token names and the associated Ada identifiers (displayed as integers). In a typical application, this tool can be used to convert an Ada constant definition package to a C header file (for resource compiler include files, etc.).



```
package Test_ID is
   BUTTON : constant := 101;
   LIST   : constant := 102;
end Test_ID;

101 43
101 42
BUTTON 6
102 43
102 42
LIST 6
Test_ID 39
Test_ID 56
parsed 0
```

Figure 9: Test package and output parse

Lessons Learned

Windows possesses a strongly C-oriented application programming interface. However, with proper Ada abstractions, many potential problems associated with language interoperability can be minimized. In addition, tools to convert representations are very useful (such as converting between Ada constant declarations and C header files). However, one limitation of the Ada language in the Windows context is the lack of a standard type-safe technique for callback functions. Until Ada 9X becomes available, this is one area that software errors will continue to occur (e.g. due to improper callback arguments).

Ada is a good choice as a base language for tool development, particularly for developing large-scale programs (such as Windows). We found that the Ada program development was not hindered by the lack of a native code debugger (thanks to the strong type checking, exception handling, etc.). However, availability of a more extensive development environment, with source browsers, etc., would be desirable.

While the Microsoft Windows GUI environment is not portable, it has the major end-user share and is becoming a *de facto* standard in the PC environment. Windows provides an environment where high-quality, interactive tools can easily communicate (using Windows DDE, DLL, object linking and embedding (OLE), etc.).

Conclusions and Recommendations

Because of the overall complexity of Windows and the effort that typically goes into coding for a GUI, a goal of the Ada developer should be to avoid manually generating Windows-specific code. Instead, reuse code through generics or automatic code generators whenever possible. For many programmers, the popularity of a language results from how few lines need to be written to run a "Hello World" program. In this regard, the use of high-level Ada constructs such as packages, generics, tasking, and exceptions provide for a productive programming environment.

A well-integrated tool architecture is not an end in itself; the environment should also include design support tools that will retain flexibility for solving a range of related problems. These tools should be affordable, have a wide distribution, and be available to designers and students. Thus, our approach is to stress tool development in a personal computer/ Windows environment which includes integrated languages, standard GUI, and provides universal device support. Using common formats for documentation and data interchange (DDE) further simplifies the integration effort.

Overall, the Windows and PC environment is recommended as a platform for tool integration as it is economical and widely used. Developers should consider developing bindings for DLLs and explore what is currently available, as this is a good way to reuse code and provide a good tool framework. The techniques described here have been used within a more extensive environment featuring several large Ada-developed tools communicating via DDE ports.

References

1. Although Ada can call other languages, Ada 9X extends this concept, see *Ada 9X Mapping Rationale*, Vol. 1, Intermetrics, Inc., March 1992.
2. Pukite, P.R., "Automated Interface Code Generation from Ada Specifications", *Ada Letters*, 13/3 (1993) p.74.
3. Nelson, M.L., and G.F. Mota, "Object Oriented Programming in Classic-Ada", *Ada Letters*, 12/2 (1992) p.102.
4. Pukite, P.R., "Ada for MS-Windows Applications", *Ada Letters*, 14/1 (1994) p.30.
5. Meridian OpenAda Windows version 2.0, Verdix Inc.

This work was supported in part by the Small Business Innovation Research program under contract NAS1-20035. The described tools are public domain and can be obtained from the author.

Software Reengineering Assessment Handbook (MIL-HDBK-SRAH)

John Clark, COMPTEK Federal Systems, Inc.
John Donald, Air Force Cost Analysis Agency
Barry Stevens, COMPTEK Federal Systems, Inc.
Sherry Stukes, Management Consulting & Research, Inc.

February 1994

ABSTRACT

Legacy software is a valuable DoD asset which should be leveraged to the greatest degree possible. There is an immediate need for DoD guidance for conducting technical, economic, and management analyses to determine when reengineering techniques are beneficial to conduct. Such guidance has been developed under the auspices of the Joint Logistics Command and the U.S. Air Force. This paper introduces the key concepts of the new draft Software Reengineering Assessment Handbook (MIL-HDBK-SRAH) which defines a process for conducting an effective technical, economic, and management assessment to determine whether and how to reengineer legacy software.

INTRODUCTION

The decision to reengineer legacy software is based on a number of technical, economic, and management factors. The new draft Software Reengineering Assessment Handbook (MIL-HDBK-SRAH) organizes these factors into a decision-making process which is targeted for the following two cases:

- A single software program needs to be assessed to determine if, and how, it should be reengineered. The SRAH process develops a recommended reengineering strategy for that program, if warranted, and an estimated return on investment and breakeven point for that strategy.

- A set of software programs within the organization needs to be assessed to determine which, if any, should be reengineered and which reengineering strategy(ies) should be used. The SRAH process develops a recommended strategy for each program, if warranted, and prioritizes the list of programs based on the need to reengineer, estimated return on investment, the estimated breakeven point, and management considerations.

The SRAH process is depicted in Figure 1 and consists of technical, economic, and management decision processes. The SRAH process begins with the identification of programs within the organization and the application of a quick screening filter to remove programs from consideration which are least likely to show a return on investment through reengineering. The resulting list of candidate programs (or, in the first case above, the single program) is subjected to a technical assessment process including a set of technical questions designed to disclose the need to reengineer. The responses to the question set are used to determine the need to reengineer and to develop a set of candidate strategies for that program.

An economic assessment process is used to determine the preferred strategies for each program. For each strategy, comparative total life cycle cost estimates are developed. Parametric cost estimating models may be used to estimate reengineering, redevelopment, and continued maintenance/support costs. Cost risk and parameter sensitivity assessments should be considered.

A management decision process is used to select the single recommended strategy for each program and to prioritize the list of programs. Selection and prioritization are based on the composite results of the technical and economic assessments and management considerations. Evaluation of the need to reengineer, return on investment, breakeven point, and management objectives and constraints results in the recommended course of action.

IDENTIFY POTENTIAL PROGRAMS FOR REENGINEERING

When evaluating a set of programs within the organization for reengineering, the MIL-HDBK-SRAH process begins with listing the potential programs for reengineering. This list could contain

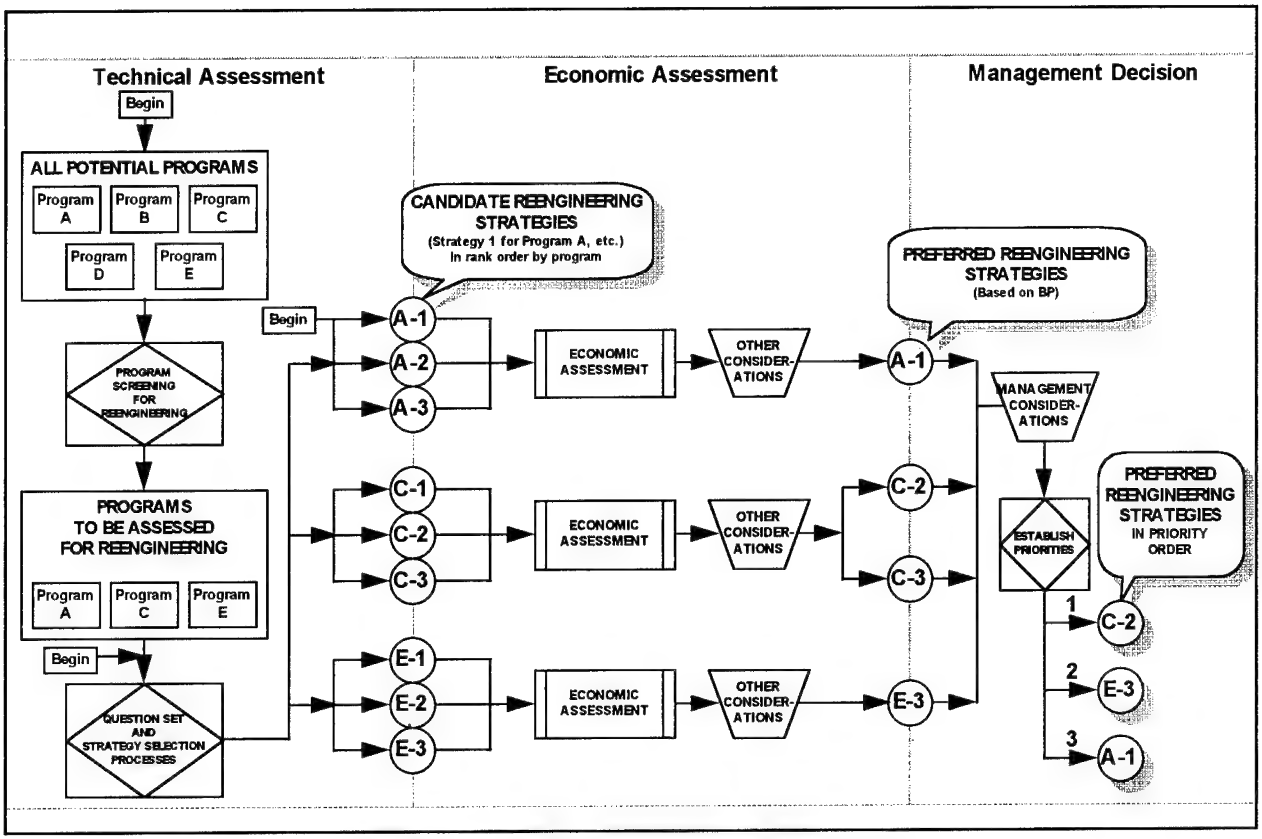

Technical Assessment
Economic Assessment
Management Decision
Begin
ALL POTENTIAL PROGRAMS
Program A
Program B
Program C
Program D
Program E
PROGRAM SCREENING FOR REENGINEERING
PROGRAMS TO BE ASSESSED FOR REENGINEERING
Program A
Program C
Program E
Begin
QUESTION SET AND STRATEGY SELECTION PROCESSES
Begin
CANDIDATE REENGINEERING STRATEGIES
(Strategy 1 for Program A, etc.)
In rank order by program
A-1
A-2
A-3
C-1
C-2
C-3
E-1
E-2
E-3
ECONOMIC ASSESSMENT
OTHER CONSIDER-ATIONS
PREFERRED REENGINEERING STRATEGIES
(Based on BP)
MANAGEMENT CONSIDER-ATIONS
ESTABLISH PRIORITIES
PREFERRED REENGINEERING STRATEGIES IN PRIORITY ORDER
1
2
3

Figure 1 - MIL-HDBK-SRAH Process

all the programs in the organization, but should at least contain any program which is perceived to be a "problem." Programs over five years old, programs which require over two manyears/year to maintain, and programs whose remaining life is over three years should be on the list.

SCREEN PROGRAMS FOR REENGINEERING

Not all of the programs on the list need to be subjected to the MIL-HDBK-SRAH process. A quick screening filter is suggested as a means to remove programs from consideration which are least likely to show a return on investment through reengineering. Programs which meet any of the following criteria may be eliminated from the list of candidates:

(1) If the program age is less than five years since Initial Operational Capability (IOC) or program deployment;

(2) If the annual maintenance budget for software is less than $250,000 per year or less than two manyears per year; or

(3) If the remaining life is less than three years.

Programs which meet criterion (1), (2), or (3) may be retained on the list if desired. Each organization may desire to develop its own criteria

REENGINEERING QUESTIONNAIRE

The need to reengineer is viewed as a function of the following system and organizational factors:

A. Size (Source Lines of Code or Function Points)
B. Complexity (Logical and Data Complexity)
C. Language Type and Portability
D. Development Process Used
E. Program Quality Trend
F. System Age
G. Level of Maintenance Activity
H. Documentation State
I. Impact of System Failure
J. Personnel Factors.

Twenty-five questions are answered for each candidate program. The questions are designed to disclose the need to reengineer. Figure 2 shows the

SYSTEM FACTORS

Factor A - Size

1. *SLOC.* How many Source Lines of Code (SLOC) exist in the candidate software?
   2-Less than 15K   4-Between 15K and 100K   6-More than 100K
2. *Function Points.* How many function points exist in the candidate software?
   2-Less than 500   4-500 to 2500   6-More than 2500

Factor B - Complexity

3. *Average SLOC/CSU.* What is the average number of Source Lines of Code per Computer Software Unit (CSU) in the candidate software?
   3-Less than 50   6-50 to 200   9-More than 200
4. *Average Cyclomatic Complexity.* What is the statistical average of the Cyclomatic complexity per module?
   3-Ten or less   6-Between 10 and 20   9-More than 20
5. *Average Essential Complexity.* What is the statistical average of the Essential complexity per module?
   3-Five or less   6-Between 5 and 10   9-More than 10
6. *Data Complexity.* Which response best characterizes the state of system data?
   3-Data is managed by relational database(s) which is/are in at least 3rd Normal Form or an object-oriented database. The data dictionary is current. Data relationships are clearly documented. A high degree of data name rationalization exists.
   6-Data is managed by relational database(s) in less than 3rd Normal Form. The data dictionary and data relationship documentation (such as Entity-Relation diagrams) are mostly current; some catch-up may be required to support major enhancement. Some cryptic names for relation tables or columns exist.
   9-Data is managed as flat files. Some data is no longer used by the program, although some effort would be required to determine which. Usable documentation of data definitions and logical relationships do not exist.

Factor C - Language

7. *4GL/3GL/Assembler.* What is the system's principal language level?
   2-4GL   4-3GL   6-2GL/Assembly Language
8. *Number of Languages.* How many programming languages does the system use?
   2-One   4-Two   6-More than two
9. *Language Portability.* Which situation best characterizes the need to change source languages?
   2-No need because: An approved HOL is being used, adequate software support tools (compilers, etc.) exist, and the implementation language permits adequate selection from a number of host processors.
   4-Some need. One of the conditions for answer #1 is not true. There is some resulting motivation to change languages.
   6-Strong need. More than one of the conditions for #1 is not true. Continued software evolution is being constrained by the current language.

Factor D - Development Strategy Discipline

10. *Development Process Followed?* The system was created using a development process that was:
    2-Rigidly followed   4-Sometimes followed   6-Not followed
11. *Change Control Discipline Followed?* The change control procedure for the system has been:
    2-Strictly enforced   4-Loosely enforced   6-Ad hoc or does not exist

Figure 2 - MIL-HDBK-SRAH Question Set

Factor E - Quality Trend

12. *Number of Errors Trend.* Over the last 6 months, has the number of errors:
    3-Decreased 6-Leveled off 9-Increased
13. *Maintenance Backlog.* Does the system have a maintenance backlog?
    3-No 6-Yes, but steady or decreasing 9-Yes and increasing
14. *Perceived Quality Trend by Customer/User.* Do the system's users think the system quality is:
    3-Improving 6-Remaining the same 9-Declining

Factor F - System Age

15. *Age Since First Release.* What is the system's age as measured from the first release?
    1-Less than 2 years 2-2 to 7 years 3-More than 7 years

Factor G - Level of Maintenance Activity

16. *Annual Change Traffic (Past 12 Months).* What is the annual change traffic of the system within the past 12 months?
    2-Less than 5% 4-5 to 10% 6-More than 10%
17. *Number of Change Releases (Past 5 Years).* How many change releases have been released within the last 5 years or since the last reengineering effort, whichever is less?
    2-Less than 5 4-5 to 10 6-More than 10

Factor H - Documentation

18. *State of Documentation.* The system's documentation is best characterized as:
    3-Complete and current
    6-Mostly complete and mostly current
    9-Non-existent or inaccurate

ORGANIZATIONAL FACTORS

Factor I - Impact of System Failure

19. *Effect of System Failure.* If the system failed what would be the effect?
    3-Little or no damage 6-Significant damage 9-Permanent damage, major financial loss, or potential loss of life
20. *Contingency Plan.* Is there a contingency plan (current, recently tested, and ready at a moment's notice) which could be used if the system fails?
    3-Yes or not needed
    6-Yes, but with some difficulty and significant loss of efficiency
    9-No
21. *Contribution to User's Mission.* How much does the candidate software contribute to the using organization's mission?
    3-Not at all 6-Small percentage 9-Significant percentage

Factor J - Personnel

22. *Percent of Maintenance Personnel With In-depth System Knowledge.* For the staff currently maintaining the system: What percentage of the maintenance personnel have in-depth experience with the system?
    3-More than 30% 6-5 to 30% 9-Less than 5%
23. *Maintainer Experience.* What is the average number of years experience as a maintenance programmer for those who maintain the existing system?
    3-More than 5 years 6-2 to 5 years 9-Less than 2 years

Figure 2 - MIL-HDBK-SRAH Question Set (Cont.)

24. *Maintainer Turnover.* What is the percentage of maintenance personnel turnover per year?
3-Less than 5% 6-Between 5 and 30% 9-More than 30%

25. *Original Developers Available?* Are the original developers available for consultation?
3-Yes
6-Yes, but the system is over 5 years old, or the original developers are not easily accessible
9-No

Figure 2 - MIL-HDBK-SRAH Question Set (Cont.)

question set. In MIL-HDBK-SRAH, a questionnaire form is provided for marking responses for each question. The average response for each factor is recorded. The resulting points are totaled to provide a quantitative result representing the need to reengineer. Programs are then rank-ordered by their need to reengineer.

The question set grew out of efforts at the Software Technology Support Center. In [STSC92], Chris Sittenauer and Mike Olsem introduced a set of questions to assist in determining if a program needed to be reengineered. Minor modifications occurred during the process of gaining technical consensus among panel members during [SB-I] and in the following months. As with the entire SRAH process, a wider consensus on the question set is now being sought.

IDENTIFY REENGINEERING STRATEGIES

Now that the candidate programs with the greatest need to be reengineered have been identified, candidate reengineering strategies can be generated for those programs. The Reengineering Strategy Selection Matrix, shown in Figure 3, identifies candidate strategies based on the responses made to the question set in the previous section.

MIL-HDBK-SRAH contains the definition of each strategy. It also provides guidance for combining strategies when more than one is indicated by the matrix and guidance for limiting the number of programs input to the economic assessment process.

REENGINEERING STRATEGIES	RESULTS OF QUESTION SET	
	Consider if ...	Probably need not consider if ...
Status Quo - Continue maintenance of existing system	Always consider	▨
Redocument	Documentation State Factor ≥ 6	Documentation State Factor < 6
Source code translation	Language Factor ≥ 4	Language Factor < 4
Data translation	Data Complexity Question ≥ 6	Data Complexity Question < 6
Restructure	Complexity Factor ≥ 6	Complexity Factor < 6
Reverse, then forward	System Factors ≥ 36	System Factors < 36
Redevelop design and code from existing requirements	System Factors ≥ 36 and remaining system life ≥ 5 years	System Factors < 36 or remaining system life < 5 years

Figure 3 - Reengineering Strategy Selection Matrix

ECONOMIC ASSESSMENT

At this point in the MIL-HDBK-SRAH process, a set of candidate reengineering strategies has been identified for each program being considered. The remainder of the process is designed to perform an economic assessment of the strategies and to execute a management decision process to select one strategy per program and prioritize the list of programs.

The purpose of the economic assessment process is to provide accurate, traceable, and credible comparative cost information, in a consistent format for each program, to allow rank ordering of the candidate strategies according to breakeven point (BP) and return on investment (ROI).

The economic assessment approach is direct, repeatable, and logical. Each program is analyzed separately and independently. For each program, the candidate strategies are evaluated by estimating total remaining life cost (RLC), which is the sum of investment (development) and support costs for the defined remaining lifetime of the software. For reengineering strategies, RLC also includes the cost of supporting the legacy software during the reengineering development time. All estimates for each program will be based on the same parametric model and the same general assumptions to allow the results to be compared. The preferred strategies are defined as those strategies whose breakeven point is less than the remaining life of the legacy software.

COST ELEMENT STRUCTURE (CES)

Each economic assessment will first establish a CES to be used for the analysis. The CES may be similar to that shown in Figure 4 or may be tailored to satisfy any of the following requirements:

(1) To match the CES from a useful and earlier estimate for this program,

(2) To match the CES output from the parametric cost estimating model used (the situation experienced in the above example), or

(3) To highlight a particular cost sensitivity, e.g., to provide more granularity into the maintenance estimate.

Since the economic assessment is a comparative process, it is not necessary to establish an exhaustive CES, particularly where the cost of the elements would be the same or similar for all strategies (facilities or utilities, for example). Costs may be estimated at the CSU, CSC, or CSCI level and aggregated to program level.

MIL-HDBK-SRAH leads the analyst through a process to consider applicable ground rules and assumptions which need to be documented with the results of the assessment.

MODEL SELECTION AND DATA GATHERING

Software estimates are normally made by level of effort, analogy, or parametric methods. For the handbook, only parametric methods (cost models) are considered. Examples using several parametric models are provided as Appendices in MIL-HDBK-SRAH. Reasons for selecting a particular model may be familiarity, availability of the model or particular input data, suitability of the model CES, or a desire to be comparable with an earlier estimate. In any case, the same model should be used for all estimates to be compared. General information on model estimating and model calibration may be found in source documents listed in SRAH.

The following list identifies the models which have been identified for use in the SRAH economic assessment process:

- COCOMO
- PRICE S
- REVIC
- SEER-SEM
- SLIM
- SOFTCOST

A Reengineering Size Adjustment (RESIZE) model is included in SRAH as an Appendix and describes a method of adjusting source lines of code for a reengineering project for input to a cost model.

ECONOMIC INDICATORS

The economic assessment of candidate strategies for each program depends on estimating and comparing economic indicators (RLC, RLC Savings, ROI, and BP) for each strategy, using the same estimating model. Figure 5 is an illustrative example of a summary of the economic assessment of five strategies for a single program. Strategy #1,

1.0 Investment (Development)
1.1 Software Development
1.1.1 Requirements Analysis
1.1.2 Preliminary Design
1.1.3 Detailed Design
1.1.4 CSU Code and Test
1.1.5 CSC Integration and Test
1.1.6 CSCI Testing
1.1.7 System Integration and Test
1.1.8 Operational Test & Evaluation
1.1.9 Site preparation
1.1.10 Development Tools
1.1.11 Program Management
1.1.12 Documentation
1.2 Site Preparation
1.3 Training
1.4 Development Tools
1.5 Hardware Development
2.0 Support
2.1 Reengineered Software
2.1.1 Software Maintenance
2.1.2 System Operations
2.1.3 Hardware Maintenance
2.1.4 Training
2.2 Legacy Software
2.2.1 Software Maintenance
2.2.2 System Operations
2.2.3 Hardware Maintenance
2.2.4 Training

Figure 4 - Generic CES

continued maintenance (Maintain Status Quo) of the legacy software, is the baseline against which the other strategies are compared. Each of the three reengineering strategies (#2, #3, #4) would require an investment (development cost) to achieve the savings shown. The final strategy (#5) would require full replacement of the legacy software and would incur the greatest investment of all candidate strategies.

The RLC Savings is defined as the RLC of Strategy #1 minus the RLC of the strategy under consideration. The breakeven point is defined as that point in time when the RLC of a strategy equals the RLC of Strategy #1, i.e., the cost of reengineering equals the cost of continuing to maintain status quo. A detailed version of the worksheet is contained in MIL-HDBK-SRAH.

In this example, the Cost Element Structure (CES) was summarized (rolled-up). Risk and sensitivity assessments were excluded from this example for simplicity. The SRAH process provides a detailed method for including risk and sensitivity assessments into the economic analysis.

From the illustrative example in Figure 5, the following conclusions can be drawn:

- Strategy #1 (Maintain Status Quo) is a low risk candidate (no investment), but its ROI (arbitrarily set at zero) places it lower than the preferred candidates. All other strategy RLCs will be compared to its RLC.

- Strategy #2 (Redocument) has the lowest investment of all strategies, but only moderate RLC Savings (as compared to #1) over the remaining life. The ROI ranking is the same as the BP ranking.

- Strategy #3 (Translate) is a poor choice, demonstrating a higher RLC than #1, the highest investment, and negative ROI values. Note also that the BP (12.8) years exceeds the remaining life (12 years). This is not a preferred strategy.

- Strategy #4 (Restructure) is clearly the preferred choice, showing the best ROI and the earliest BP.

Program: Illustrative Case FY94 $K	Candidate Strategies				
	Maintain Status Quo	Redocument	Translate Source Code	Restructure	Redevelop
<u>Parameters</u>					
SLOC	31,574	31,574	31,574	31,574	31,574
Design Modified (DM)	0%	0%	0%	10%	50%
Code Modified (CM)	0%	0%	100%	20%	80%
Integ & Test Modified (IM)	0%	0%	100%	100%	100%
New Documentation (ND)	0%	100%	20%	50%	80%
Equiv SLOC (ESLOC)	0	6,946	19,386	14,556	24,975
Annual Chg Traffic (ACT)	20%	19%	18%	16%	15%
Remaining Years (Y)	12.0	12.0	12.0	12.0	12.0
Reengineering Years (YR)	0.0	0.6	0.9	0.8	1
Support Years (YS)	12.0	11.4	11.1	11.2	11.0
<u>Cost by CES</u>					
1.0 Investment	$0	$344	$1,179	$836	$1,597
2.0 Support	$4,232	$3,650	$3,136	$2,496	$2,098
2.1 Reengineered Software	$0	$3,435	$2,818	$2,210	$1,746
2.2 Legacy Software	$4,232	$215	$319	$286	$351
<u>Economic Indicators</u>					
1. Remaining Life Cost (RLC)	$4,232	$3,994	$4,315	$3,331	$3,695
2. RLC Savings vs #1	$0	$239	($83)	$901	$537
3. Return On Investment (ROI)					
a. Investment ROI (ROI_I)	0	0.69	(0.07)	1.08	0.34
Rank Order	4	2	5	1	3
b. Total ROI (ROI_T)	0	0.06	(0.02)	0.27	0.15
Rank Order	4	3	5	1	2
4. Annual Support (AS)	$353	$301	$254	$197	$159
5. Annual Support Savings	$0	$52	$99	$156	$194
6. Breakeven Point (BP) in Years	12.0	7.2	12.8	6.2	8.2
Rank Order	4	2	5	1	3
7. Preferred Reengineering Strategy	N/A	Yes	No	Yes	Yes

Figure 5 - Summary Comparison Worksheet for a Single Program (Example)

- Strategy #5 (Redevelop) is clearly the greatest investment, provides only modest RLC Savings, but still displays a BP within the remaining life. It may also incur the greatest cost and schedule risk, something that should be investigated in accordance with procedures described further in MIL-HDBK-SRAH.

Ranking of strategies is assigned in ascending order of BP or ROI, with the preferred strategies being #4, #5, #2, and #1 in that order.

COST RISK AND SENSITIVITY

MIL-HDBK-SRAH includes a process for reevaluating the results of the economic assessment and assessing the associated cost risks and parameter sensitivities. Guidance for documenting the economic assessment process is provided.

MANAGEMENT DECISION PROCESS

At this point in the MIL-HDBK-SRAH process, a set of preferred strategies has been identified for each program being considered. The remainder of the process is to select one strategy per program and prioritize the list of programs. Selection and prioritization are based on the composite results of the technical and economic assessments and management considerations. Evaluation of the need to reengineer, economic indicators, and management objectives and constraints results in the recommended course of action.

While the technical and economic assessment processes are well defined in SRAH, the management decision process is more sensitive to less well defined and less tangible program considerations and is based primarily on judgement. Non-quantifyable program considerations include perceived risk, resource availability, requirements realism, estimate credibility, and schedule uncertainty.

The primary objectives of the management decision process are to reduce software support cost, improve software quality, and meet other management or organizational objectives.

The ideal choice for the highest priority program would be if all of the following were true:

- Greatest technical need to reengineer
- Earliest BP within the remaining life
- Highest ROI
- Investment cost within budget
- High confidence in estimate and schedule
- Remaining life confirmed
- Legacy software has the highest support cost
- High probablilty of project success.

HISTORY OF MIL-HDBK-SRAH

The Joint Logistics Commanders (JLC) Joint Policy and Coordinating Group (JPCG) on Computer Resources Management (CRM) initiated the first draft of this handbook at the First Software Reengineering Workshop [SB-I] in September 1992. The handbook was developed by the members of the Reengineering Economics Panel at the workshop and was entitled *Reengineering Economics Handbook (MIL-HDBK-REH)*. In April 1993, Comptek Federal Systems, Inc. and Management Consulting & Research, Inc., refined and enhanced the handbook under contract to the Air Force Cost Analysis Agency. Refinement and use of MIL-HDBK-SRAH continues under Joint Logistics Command and Air Force Cost Analysis Agency direction.

FOR FURTHER INFORMATION

Further information regarding MIL-HDBK-SRAH may be obtained through the Air Force Cost Analysis Agency (AFCAA) from Mr. John B. Donald, AFCAA, 1111 Jefferson Davis Hwy, Suite 403, Arlington, VA 22202; phone: (703)746-5865 or (703)692-0006; MILNET: donald@afcost.af.mil. Copies of SRAH may be obtained from the Air Force Software Technology Support Center (STSC), Hill AFB, Utah; phone: (801)777-8045; or Mr. Chris Sittenauer at the STSC; phone: (801)777-9730. Comments regarding the handbook are particularly solicited and should be sent to Mr. John Clark, COMPTEK Federal Systems, Inc., 2877 Guardian Lane, Va Beach, VA, 23452; phone: (804)463-8500; clark@comptek.mhs.compuserve.com..

REFERENCES

[SB-I] *Workshop Proceedings*, JLC-JPCG-CRM First Software Reengineering Workshop, Santa Barbara I, 21-25 September 1992

[STSC92] Chris Sittenauer and Mike Olsem, "Time to Reengineer?" *CrossTalk*, Issue 32, March 1992.

APPLICATION-SPECIFIC LANGUAGES AND ADA: ARE THEY COMPATIBLE?

Christine Braun
GTE Federal Systems
15000 Conference Center Drive
Chantilly, VA 22021
(703) 818-4475

Introduction and Background

In many application domains we are beginning to see a move away from general-purpose procedural languages (sometimes called "third generation languages") toward higher-level application-specific languages (sometimes called "fourth generation languages" or "4GLs"). Ada is an example of a third generation language (though it is one of the most powerful and flexible ones). Application-specific languages include the MIS-oriented languages supported by popular CASE tools, as well as other languages designed to fit specific application domain needs. Most application-specific languages are processed by programs that generate third-generation languages; that is, they convert the program to a third-generation language. These processors are often called "application generators" or "program generators".

In the areas for which they have been developed, application-specific languages show impressive programming productivity gains. Improvements of one to two orders of magnitude are not uncommon. There is a strong argument to be made for use of these languages when a suitable one is available.

On the other hand, such languages lose one of the principal benefits organizations seek with Ada—the use of a single common language. By their very nature, application-specific languages cannot be common to a large, heterogeneous body of software. There are many significant arguments for a common language:

- Maintainers need only learn and use one language.
- Fewer distinct development and maintenance tools are required.
- Training needs are reduced.
- Those who review software and documentation (customers, managers, QA personnel, etc.) have a common basis for understanding.
- There are more organization-wide reuse opportunities.
- Software is more likely to be portable to other platforms.

All of these reasons contribute to cost savings.

Today the DoD mandates the use of Ada. Does this mean application-specific languages are forbidden? Or is the generated language what counts? Is the application generator simply a CASE tool that helps produce Ada code more quickly? If so, what is maintained—the application-specific language code or the Ada? If it's the Ada, isn't that losing most of the benefits of the higher-level language? If it's the higher-level language, isn't that losing the maintenance benefits noted above?

Many software developers today are wrestling with these questions. They wonder what makes sense for their organizations and customers, and whether there is a good middle ground. This paper will explore these issues, present different perspectives, and propose alternative approaches.

Application-Specific Languages

What are application-specific languages? Application-specific languages are programming languages that allow software developers to create software application programs in a much higher-level language tailored to the application domain. The languages contain constructs that are particular to the domain. A language for developing MIS applications might provide powerful capabilities for interacting with a database management system or data dictionary. A language for designing screen interfaces might provide high-level constructs for placing information on a screen and extracting information entered on the screen by the application user.

Perhaps the best known application-specific languages are Fourth Generation Languages (4GLs), designed for creating DBMS-oriented information system applications. Because 4GLs focus on a narrow class of applications, they can include very powerful constructs that allow software to be developed quickly and easily by those familiar with the application domain.

Application-specific languages can be (and have been) developed for other types of applications as well. They are best suited to narrow domains, or subdomains of large domains such as Command and Control [1]. Because they require a domain specific vocabulary for expressing applications, they are generally unique to the domain or subdomain and not easily modified to handle other domains. Creation of an application generator for a particular domain, furthermore, is a significant undertaking. Development of an application generator is most appropriate in domains that are well-understood and in which many different developments perform primarily the same kinds of processing.

In the term "application-specific language", the noun "language" must be interpreted as that combination of syntactic constructs used to completely specify the application. Often application-specific languages are not strictly textual in nature. Languages for creating interactive information systems, for example, often provide explicit on-screen support for laying out screens and forms. Some application-specific languages provide graphical support for program development. Programs are actually written (perhaps partially) using graphic forms such as data flow diagrams, object diagrams, etc. Programming in such languages does not simply involve writing lines of ASCII source and then submitting it for processing; it is a much more interactive process. Further, the boundary between the "language" and the tools that process it is not as well defined. Programmers create programs using tools, developing specifications in some kind of higher-level syntax that perhaps combines text and graphics, and perhaps explicitly or implicitly incorporating library routines that perform many functions common to the application domain. The end result is an executing program, but the process of developing it is not necessarily "programming" in quite the traditional sense.

The adjective "application-specific" can also be somewhat misleading. Many of these languages/tools support creation of a wide variation of applications, typically by offering a rich subprogram library. Alternative libraries for the same product may focus on different kinds of applications, or products may focus on what we call "horizontal" domains—addressing a particular kind of processing (e.g., graphic user interfaces) across applications.

As a brief illustration of the available application-specific language technology, several examples are:

Texas Instruments' Information Engineering Facility (IEF) [4]. Using IEF, the application is specified using a combination of logic diagrams and interactively-developed textual specifications. The user interface provides menus to help specify applications in terms of the data model, e.g., to help create database queries and updates. Tools also support on-line testing, integrating the work of multiple developers, etc. Figure 1 is an example of a partial IEF specification.

```
READ organization
    WHERE DESIRED org cost_center = import org cost
    center
    AND DESIRED org division = import org division
WHEN successful
    READ job_grade
          WHERE DESIRED job_grade number = import
          job_grade number
    WHEN successful
          CREATE employee
          ASSOCIATE WITH org WHICH contains IT
          ASSOCIATE WITH job_grade WHICH
          is_assigned_to IT
          SET number TO import employee number
          SET name TO import employee name
          SET rate TO import employee rate
          SET direct_indirect TO import employee
          direct_indirect
```

Figure 1. IEF Example

Template Software's SNAP [3] is a toolset that supports the development of object-oriented applications. Development begins with creation of the application's object structure. SNAP then allows programs to be specified in terms of the object attributes and relations, and the names and interfaces of their operations. The algorithms for the operations are then specified in a high-level specification language, as illustrated in Figure 2. Many library routines are provided for use in creating these operations.

```
CreateDescriptor:  public own member of DESCP
            (atrib nam: in str,
             MFC: in str,
             valid func: in str)
     descp mbr: member of DESCP

/* check to see if member already exists */

     descp mbr := UTIL::IsMem("DESCP", atrib nam).

/* if member doesn't exist add and initialize attributes */

     if status(descp mbr) = unknown then
            descp mbr := UTIL::CreateMember("DESCP",
                                   atrib nam).
            \addvalue mymember>Descps, descp mbr.
            \AddStr(mymember)>Attribute Lst, atrib nam).
            descp mbr>MFC := MFC.
            descp mbr>Attribute Name := atrib nam.
            descp mbr>Validation Function := valid func.
     endif.

     return descp mbr.

endfunction.
```

Figure 2. SNAP Example

The DSSA Automated Message Handling Language [1] is a capability developed under the ARPA Domain Specific Software Architectures program illustrating the use of application generation technology in the Command and Control domain. Built on the AP5 technology developed by the University of Southern California Information Sciences Institute, this language supports the high-level specification of the parsing and syntax specification rules to be applied to a military formatted message. Figure 3 is an example.

AdaSAGE [2] is a toolset supporting the creation of Ada applications, with particular support for information systems applications. It provides interactive support for creation of user interfaces, for developing a data dictionary describing the structure of the application's data, and for specifying applications in terms of that data structure. As a public domain capability, it has been widely adopted in the Ada community.

```
type SPOT = (FORCE), (SHIPTK | AIRTK | AIRCRAFT),
            SHIP
     validations
            disallow MSGID.message-serial-number;
            require SHIP.location
            no SHIPTK and no AIRTK requires FORCE;

dataset MSGID = message-code-name (originator)
                 (message-serial-number) (as-of-month)
                 (as-of-year) (as-of-DTG)
     validations
            as-of-DTG precludes as-of-month;
            as-of-DTG precludes as-of-year;
            as-of-year requires as-of-month;
            message-code-name /= SPOT requires
                originator;
            message-serial-number and no as-of-DTG
                requires as-of-month;
            field message-code-name = A*26;
            field originator = A*25;
            field message-serial-number = N 3;
            field as-of-month = month;
            field as-of-year = N 4;
            field as-of-DGT - day, hour, minute, (Z), SUM1,
                month,  year;
```

Figure 3. Message Specification Language Example

How are application-specific languages processed? The tools associated with application-specific languages provide the capability to produce an executable program from the specification provided by the application developer. This can be done in several ways, often in combination:

- The program written in the specification language can be translated into an equivalent program in a lower-level (third generation) language, thus "generating applications." The generated code is then compiled through a compiler for the lower-level language.
- The program written in the specification language can be translated into an internal form—e.g., tables or rules—that is executed by a runtime component provided as part of the toolset.
- The executable program can be constructed of calls to library routines, with "glue" code generated to connect them. Most of the

application variability can be handled by parameters to the library routines. (For example, this is the way DBMSs are incorporated into applications.)

Of the examples given in the preceding section: IEF translates into a target third generation language (COBOL and C are available; Ada is in development), with library routines incorporated; SNAP produces tables that are executed by a runtime engine, linking with library routines written in C; the DSSA Message Handling Language is translated to Ada with calls to library components for database access; and AdaSAGE produces executable tables and links in library routines written in Ada.

How are application-specific language programs maintained? The benefits of application-specific languages come from their enhanced expressive power and from the use of the associated tools that eliminate many sources of potential programming errors. These same benefits are available during maintenance, but they are available only if the software is maintained at the specification language level. An application originally written in a high-level language with knowledge of the system's data organization, etc., is clearly best maintained in that same form.

What about the alternative—maintaining the generated form of the program? As noted above, this generated form can take several forms, but it typically includes generated lower-level language code and/or an internal tabular representation of the program. There are three major drawbacks to maintaining the program at this level:

- As noted above, the advantages of productivity and error-avoidance offered by the application-specific language tools are lost.
- Changes to the generated form of a component will eliminate its correspondence to its specification, thus potentially invalidating all data maintained by the toolset relative to the entire application. Typically recovery from this state is not possible.
- The generated form is not designed for maintenance by humans; it does not have the structure and readability considered essential in maintainable software.

Application-specific languages and reuse. With the increasing focus on software reuse, application-specific languages/tools are being recognized as a major reuse mechanism. The relationship between this technology and reuse is apparent on several levels:

- The earlier discussion noted that processing of the application-specific language programs often involves the incorporation of library routines. These are all reusable components; they are reused in the constructed application. (The fact that they may be perceived as language elements by the programmer does not change this.)
- Because components written using an application-specific language are easily understood, modified, and maintained, they are in fact prime candidates for reusable components themselves.
- Reuse professionals have noted that the greatest benefits come from identifying and introducing reuse as early as possible in the system life cycle, ideally beginning at the requirements definition level. By supporting the creation of reusable specifications, this technology is a step in that direction.
- The reuse community is increasingly recognizing that simply providing component libraries is not going to achieve the full promise of reuse. The current focus on domain specific software architectures is based on the premise that reuse can be furthered by providing specific application architectures or frameworks in which programmers can "plug in" reusable components with standard compatible interfaces. In fact, the kinds of tools described here provide exactly such frameworks.

Issues and Tradeoffs

The greatest advantage to the use of application-specific languages is greatly increased productivity. Research has shown that the time programmers take to produce code is essentially proportional to the number of lines of code written. It makes little difference whether those lines are assembly language, Fortran, Ada, or a fourth-generation specification language. However, these languages cover a wide range of expressive power. Ten lines of Ada can do far more than ten lines of assembly language; the ten lines might compile to 50–100 lines of machine-level code. Similarly, ten lines of a fourth-generation language might be equivalent to 50–100 lines of a third-generation language (particularly if invoked reusable library components are considered). But each ten lines takes about the same level of programming effort. Thus, application-specific languages can increase productivity by one to two orders of magnitude. (It is worth noting, though, that such increases are not always obtained; they depend on a good fit between the language and the problem. If the program takes little advantage of the power of the language, and is not able to use reusable library resources, benefits are reduced.)

The preceding paragraph focuses on coding productivity, since that is relatively easy to measure, but the productivity benefits of application-specific language use are apparent in other phases. The requirements analysis phase can be simplified by expressing requirements in the terminology of the application-specific language. In the design phase, the application-specific language and tools provide, in effect, a design framework and a set of reusable application constructs. These larger design building blocks expedite design in a manner similar to the coding benefits. Testing is also simplified; there is less code to test, it is easier to understand and debug, and toolsets typically include language-level test support tools. Maintenance, of course, takes advantage of these same savings. Thus, life-cycle costs can be reduced by an order of magnitude or more.

Another major benefit of an application-specific language approach is software quality; software is more readable, maintainable, and robust. This approach eliminates many opportunities for programmers to make errors. It provides a framework for development that helps ensure that good design principles are followed, that no functions or checks are overlooked, and that data mismatches are avoided. Further, it provides reusable elements that have already been thoroughly tested. Applications are not only developed more quickly; they are more reliable throughout their lifetime.

Clearly, there are many benefits to using application-specific languages. There is also a potential downside—language proliferation. The major objective of the DoD's common high-order language initiative (that led to the development of Ada) was not a good high-order language—there were already several—but commonality. The proliferation of languages used in the systems maintained by the DoD created a maintenance nightmare. Maintainers had to work with many different languages and with a wide range of supporting methods and tools. It also increased development cost. Training requirements were increased, tools proliferated, and reuse was greatly inhibited. By standardizing on a common language the DoD created a foundation for building a strong development/maintenance skill base, for acquiring tools with wide applicability, and for establishing a far-reaching reuse program.

Application-specific languages threaten this position. There are many of them, with more in development. If the DoD, for example, were to begin acquiring systems that used just the four described earlier—IEF, SNAP, the DSSA Automated Message Handling Language, and AdaSAGE, the benefits noted above would be decreased. Instead of a single set of maintainer skills, there would be different training required to maintain software developed by each method. Tools and methods would also be partitioned by language. Reuse would be simple only within a single set. With these four application-specific languages added to Ada, the DoD would have five "families" of systems to support—and it would not stop at five.

This quandary is applicable not only to the DoD. Any organization can benefit from language standardization in reduced training; fewer, well-understood tools; common methods and procedures; and extensive reuse. If the organization instead selects the best application-specific language for each project, this standardization weakens.

Approaches

The following paragraphs describe several possible approaches to the dilemma described above.

Stick to a common third-generation language (Ada or another). An organization that has a heavy investment in standardization on that language (lots of existing software, well-established and understood procedures and tools, an active reuse program) might find it best simply to stick to that standard. Many of the advantages of an application-specific language can be gained through development of generic architectures and a strong reuse practice, without any actual language change.

Allow use of application-specific languages provided their processors generate Ada applications. This is an approach advocated by some groups in the DoD today. However, as noted earlier, systems developed using application-specific languages/tools can be properly maintained only by using the original development tools. Thus, the generated Ada or any other internal representation should make no difference to the maintainer. Such a position is, perhaps, just avoiding the issue, or complying with the letter of the law but not its spirit.

Allow use of tools that support specification and automated construction of Ada programs, viewing them as tools rather than alternative languages. While there is an increasingly fuzzy boundary between design tools and languages, this may also be begging the question. The real question must be "What is maintained?" As long as it is something other than an Ada source program, in effect, a higher-level language is being used.

Allow any desired use of application-specific languages considered best for the job at hand. This approach may appear to take the best advantage of evolving software development technology, but it ignores many lessons about life-cycle costs that have been learned through the proliferation of third-generation languages.

Standardize on a single application-specific language, permitting only that language and "straight" Ada in the organization. Any such standardization reduces the scope of the problem. However, is the organization produces many kinds of applications, one application-specific language is not likely to be suitable for all. Clearly, any language that meets all the needs of the DoD could not be considered an application-specific language.

Define/adopt a common approach to application-specific languages, allowing a family of such languages. Today, application-specific languages differ significantly in technology, capability, and user interface, as well as in the kinds of applications they support. It might be feasible to adopt a single technology for providing such languages—a sort of framework or generic processor—using it as the basis for constructing a family of application-specific languages with different constructs but a common operation and "look and feel." Further, these should provide the capability to construct applications using elements from a common Ada library, thus retaining the ability to reuse software across application-specific language boundaries.

Summary and Conclusions

Application-specific languages are an exciting and major step forward in software development technology. The very real productivity gains they offer cannot be overlooked. However, they are inherently in conflict with the concept of language standardization that has enabled the Ada community (and others) to achieve great benefits in maintainability and reuse. Organizations that rely on the Ada standard have adopted various views on the use of such technology; there is not yet a consensus, and answers chosen by some do not appear suitable for others.

The approaches that have been chosen run the gamut from ignoring application-specific languages to embracing them indiscriminately. No approach seems to achieve an obvious "best" balance of the benefits of the two. It is interesting, though perhaps premature, to consider the possibility of a cost model that would measure the benefits of alternative approaches for a given organization's needs.

In the 1960's and 1970's, hundreds of third-generation languages were introduced. Inventing new languages was a popular pursuit of computer science scholars. Eventually, the drawbacks of this proliferation became apparent, and developers have gradually settled on a few favorites. We can expect that higher-level, application-specific technologies will have a similar growth spurt and settling period.

However, because of their application-specific nature, standardizing on a single one does not make sense. One promising avenue for moving toward greater commonality, which requires further exploration before it can be accepted, is the notion of a common framework for application-specific languages. This could potentially allow the greater expressive power afforded by these languages while providing many of the benefits of today's Ada standardization—common methods and tool "look and feel," reduced training and maintenance needs, and reuse across application boundaries.

References

[1] Braun, Christine, W. Hatch, T. Ruegsegger, B. Balzer, M. Feather, N. Goldman, and D. Wile, "Domain Specific Software Architectures—Command and Control", *Proceedings of 1992 IEEE Symposium on Computer-Aided Control System Design*, Napa, CA, March 1992.

[2] Stewart, Howard, Kenneth Russell, and Paul Whittington, *AdaSAGE Notes*, Idaho National Engineering Laboratory, January 1993.

[3] Template Software, SNAP Marketing Literature, Herndon, VA.

[4] Texas Instruments, IEF Marketing Literature, 6550 Chase Oaks Boulevard, MS 8474, Plano, Texas 75023.

Title:

"Ada, a Software Engineering Tool, in Introductory Computer Science Programming Courses at Sacred Heart University : Mutual Benefits"

Submitted By:

Sandy Honda Adams
Sacred Heart University
5151 Park Avenue
Fairfield, Connecticut 06432-1000

To:

Ms. Marge Risinger
Rosenberg & Risinger, Inc.
11287 West Washington Blvd.
Culver City, CA 90230
(213) 397-6338

Ada, a Software Engineering Tool, in Introductory Computer Science Programming Courses at Sacred Heart University : Mutual Benefits

Sandy Honda Adams
Sacred Heart University
5151 Park Avenue
Fairfield, Connecticut 06432-1000

An Ada experience in the undergraduate curriculum has been a positive one for both students and the teacher. The programming features of Ada made the course successful! Without teaching software engineering formally to these students, the course gave the students a natural basic understanding of some of its important themes.

Sacred Heart University has offered a computer science major since fall 1983. This program offers two options to service both the business and scientific communities. The curriculum is continually updated following American Computing Machinery (ACM) curriculum guidelines. There are currently 200 computer science majors at the University. Graduates of our program are employed in banks, financial firms, software development firms, utility companies, local and state government, manufacturing firms, and commercial and defense contractors.

The University's current computer science curriculum has been modified pursuant to a small Advanced Research Projects Agency (ARPA) grant entitled "Undergraduate Curriculum and Course Development in Software Engineering and the Use of Ada", used to implement two courses using Ada. The two introductory computer science courses are CS050 (Introduction to Programming") and CS051 (Data Structures). The courses allow the student to apply Ada, its support environment, and some principles of software engineering to build software that is maintainable, efficient and understandable at an introductory level.

Both the Introduction to Programming course and the Data Structures course went better than expected, given the misconception that Ada is too difficult for introductory programming courses. I did not find Ada significantly more difficult to teach than alternative languages used before, such as Pascal and Modula2. Student reaction was very favorable. From comments on a survey given at the end of the introductory course, students indicated that they found the language "english-like", "very readable", "quite logical", "easy to learn", "painless", and one student added "fun!".

Some of the students entering the course, knowing that Ada was a programming language used by the Department of Defense, seemed skeptical of learning Ada. They assumed that any language developed and used by the Department of Defense, especially for embedded systems, must be anything except a "general purpose language".

After tasting some of the rich flavors of Ada, students read articles about commercial applications in Ada, such as the development of the air traffic control software in California, the development of banking software in the european banking community (Bank of Finland), and also in San Francisco (Wells Fargo Nikko), and about the high-profile Ada success at a Reuters Financial Services in Hauppauge, NY. Once students realized that Ada was not just a defense language and that using Ada was required for success, students quickly adopted the language. They did not find Ada a difficult language to work with on the introductory level.

One of the real "turn-ons" for the students was being introduced to the language with the history of Ada, and a sample program that simply drew a face multiple times using a for loop. (Fig.1)

The program was kept very simple with a for loop and put_lines. It allowed for the explanation of parts of an Ada procedure, the introduction to the concept of packages (text_io), and the use of one of its resources (put_line). The first assignment was to create a picture of their own design. The assignment was fun and most students tried to outperform their peers. Students created the Eiffel Tower, wrist watches, a cat watching a gold fish bowl, chemistry lab glassware, a PC with my face on the monitor portion, the Simpsons, a variety of picturesque houses, boats, cars, etc. I would electronically collect their files and create a class package and body containing all of their wonderful work of art! (Fig. 2)

I then wrote two sample programs, one which simply used their package to draw a few of their works of art. It allowed for the explaination "using" package class and all its available resources. (Fig.3) The second (Fig.4) contained a screen package, a loop around a menu of art choices, the selection of a drawing, and a case statement to run the appropriate procedure to draw the art work. From these intuitive applications, looping, input, selection, menus, procedural calls, and packages could be covered.

Fig. 1

```
------------------------------------------------------------
-- This procedure draws the three faces of Ada.
------------------------------------------------------------
with text_io;
   procedure AdaFaces is
         begin -- AdaFaces
               for i in 1..3 loop
                   text_io.put_line("      {>o<}      ");
                   text_io.put_line("    {{{{{{}}}}}}     ");
                   text_io.put_line("   {{{ ^   ^ }}}    ");
                   text_io.put_line("  {{{  o   o }}}    ");
                   text_io.put_line("  {{{    ^    }}}    ");
                   text_io.put_line("  {{{    =    }}}    ");
                   text_io.put_line("   {{         }}    ");
               end loop;
         end AdaFaces;
```

Fig. 2

```
------------------------------------------------------------
-- This package --contains CS050 student art
-- files      Fall 93
------------------------------------------------------------
package Class is
         procedure Eiffel;
         procedure SpeedBoat;
         procedure Simpsons;
         procedure CatsDelight;
         -- etc
end Class;
```

Fig. 3

```
---------------------------------------------------------------------------
-- This main procedure will use two of the resources of the class
---------------------------------------------------------------------------
        with Class;
        procedure Main is
                begin -- main
                        Class.Simpsons;        -- invoking a procedure Simpsons
                        Class.Eiffel;          -- invoking procedure Eiffel
                        Class.Simpsons;        -- invoking procedure again
                 end Main;
```

Fig. 4

```
        ----------------------------------------------------------------------
        -- This procedure will present you with a menu to select from
        --  It makes the package class available
        ----------------------------------------------------------------------
        with text_io,Class, Screen;
        procedure ArtShow is
                Choice : character;          -- variable declaration
            begin -- ArtShow
                loop
                        text_io.put_line("          Menu Selection!");
                        text_io.put_line("---------------------------------------------  ");
                        text_io.put_line("  (A)  Eiffel            (G) CatsDelight      ");
                        text_io.put_line("  (B)  Simpsons          (H) SpeedBoat        ");
                          -- etc
                        text_io.put_line("  (F)  MyHouse           (X)  To Exit Menu   ");
                        text_io.put_line("---------------------------------------------- ");
                        text_io.put("    Your choice A, B, C, ... or X please ==>   ");
                        text_io.get(Choice); new_line(2);
                        screen.clear;                                   -- clears screen
                        screen.position(row=>5,column=>10);              -- positions cursor
                        case Choice is
                                when  'A' | 'a'  => Class.Eiffel;
                                when  'B' | 'b'  => Class.Simpsons;
                                when  'C' | 'c'  => Class.CatsDelight;
                                -- etc
                                when  'X' | 'x' => exit;
                                when others   => text_io.put_line("Wrong choice dummy!");
                        end case;  new_line(3);
                end loop;
            end ArtShow;
```

The students really enjoyed seeing what others did. They were comfortable, having fun, and eager to learn. They learned that packages were reusable. That the specification gave the user useful information, but the body was not required to be seen. Two important concepts were taught in a painless manner - ***information hiding*** and ***reusability***.

The need for proper indentation was emphasized. They found the Ada code very readable. Whenever the comfort level in the classroom is high, students relax and absorb more. The students were hooked on the class, curious about Ada, and decided that they wanted to learn more. From here we were able to learn the concepts of program design, programming logic, flowcharting and top down design, as well as the basic programming statements that are included in any programming and data structure course, stressing user documentation, program design, coding, style, and aesthetics.

Software engineering principles and practice naturally become` an intrinsic and integral part of the courses. Data structures allowed for the application of subprogram development, packages, generics, and exception handling. Ada's prominent features enabled the programmer to write code that is re-useable, generic and simple. Since these are the basic goals of software engineering, Ada becomes a natural tool for it. Students in an elementary programming course can begin to appreciate a language rich in features, and understand the basic goals and principles of software engineering that evolves from coding in Ada. It is something that seems to follow naturally. Both language and discipline work in a symbiotic relationship to benefit the software developer, a concept that students can learn at an early stage in their computer science program.

Making Ada "Exciting and Fun" for Undergraduates

Akhtar Lodgher, Ph.D
Hisham Al-Haddad, Ph.D

Department of Computer Science and Software Development
Marshall University, Huntington, WV 25755
Phone: 304-696-2695, Fax: 304-696-4646
Email: lodgher@muvms6.mu.wvnet.edu

Abstract:

Programming for beginning computer science students may get boring because programming is a mundane task. Students learning Ada in the freshman course may be more susceptible to this because of the size of the language. In this paper, we present an approach by which "interesting" programs using menus and graphics can be introduced to beginning students.

Introduction

Programming is a mundane task. For the incoming freshman, computer science is viewed as nothing but programming in the language being taught. This is evident from the fact that if a freshman is asked what computer science courses he or she is taking, the answer is "Ada", or "Pascal", or "C", irrespective of names such as "Introduction to Computer Science I", etc., The student very quickly gets wrapped up in the syntax of the language, and if there are problems with the coding environment, or unavailability of labs at their convenience, or other superficial excuse, their interest in the subject quickly wanes.

Ada, no doubt, is a large language. Due to its size, it may seem especially threatening. One of the strong features about Ada is that problem solution can be taught in the context of design and implementation quite elegantly using good software engineering practices. A minor side effect of this feature is code size, which can tend to be large for a small problem. Students who have learned languages such as Pascal or C in their high school tend to question the size, and find short cuts. Their perception is that Ada can be used only for solving large system problems. Hence, to program small "neat stuff", some other language such as Basic, Pascal or C is a better choice.

The focus of this work is to present some solutions of keeping the motivation level high and for developing a positive attitude towards Ada. "Cool stuff" can be done using Ada, and once students find out how to do "cool stuff", they will be hooked to it, and spread the word. Also, we believe that "cool" examples of programs written in Ada using color, graphics, windows, sound, etc., should be demonstrated in the classroom. Students would then raise questions such as "How did you do that?" or "Can you really do this using Ada?". Their motivation can be kept high by informing them that the skills they learn will help them create programs with graphics, sound and animation. Using Ada, "cool stuff" can still be done using good software engineering practices.

In this paper, we describe our approach of integrating colors, graphics, windows, etc., in undergraduate courses. Details of a few packages, which we have implemented using our approach, are described. Even though our examples may seem a little simplistic, we believe that the approach is quite powerful. These packages help beginning students use features such as menus and graphics without knowing too many system related details. We are in the process of developing more packages for the beginning and intermediate level courses.

Our Approach

In this paper we present a mechanism of developing packages and program units which allow a software system to communicate with a user through more appealing mechanisms such as menus, graphics and windows. These packages are developed with the following overall objectives:

- Demonstrate that Ada can be easily used for small applications.
- Attract students for using the Ada language and keeping their interest and motivation high.
- Integrate the developed materials without much perturbation.
- Ease the side effects that students may encounter due to the size of the language.
- Use these materials in upper-level courses for larger applications that suit the course level.

Currently, we have developed some packages for the introductory set of courses. We believe that if students start using these packages early on, then by the time they get to the upper level courses, their curiosity would be raised enough to "dig" into the details, and unravel the secrets.

One might argue against the use of graphics, windows, and graphical user interface packages in early courses because it increases the "syntax" burden on students, on top of an already large language such as Ada. Our counter argument is that if students are shown that Ada can be used for a wider set of applications, that it is as competitive (if not more competitive) as other languages such as C, and that "neat and nifty" programs can be written in Ada, it will make Ada more interesting. This will help them to stay focused and keep their confidence level up.

The conventional approach:

The use of graphics, windows, a graphical user interface (GUI), etc, involves the use of many system dependent features. Appropriate libraries have to be accessed, and the functions available in each must be known. In the regular or conventional approach, development of a software system using graphical elements would involve the development of three packages (or units), namely: a package containing the graphical resources, a package containing the interface code and the application package. It is possible that each of these units may contain multiple packages. The relationship among these packages is shown in Figure 1.

For a GUI system such as Windows, the graphical resources package would contain all the resources such as dialog boxes, icons, etc., the interface code package would contain all the

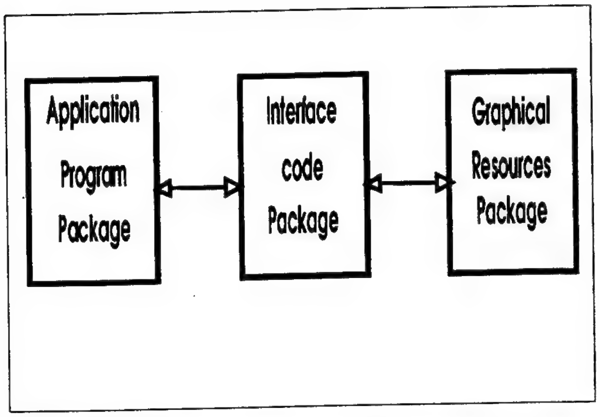


Figure 1: Package relationships using conventional approach

functions which handle the message passing between the dialog boxes, and the application package would use the functions in the interface code package. For a system without a graphical user interface, such as DOS, the graphical resource package would contain all the system available resources for drawing graphical elements such as lines, boxes, etc., the interface code package would contain functions providing a higher level of graphical elements to be used by the application. These functions would call selected functions from the graphical resources in varying sequences.

Example 1 illustrates the contents of each of these packages for a menu program. The level of abstraction of the functions in the interface code package is an important feature. As the abstraction level increases, the calls from the application programs can be made more abstract. However, as the abstraction level increases, the interface code package becomes less versatile. This means that it cannot be used for a wider set of applications.

In general, the interface code packages provide functions at a lower level of abstraction, making them more versatile. The disadvantages of this, from an academic perspective, are: *(i)* it makes the development of the interface code package more difficult because thorough knowledge of the graphical resources is required and *(ii)* it makes the design of the application packages more difficult because appropriate parameters must be passed to trigger a function in the interface code package.

Our proposed approach:

To overcome the disadvantages of the conventional approach, our approach uses an additional package called the "custom" package, between the interface code package and the application. This custom package provides higher level functions to the application package by making appropriate lower level calls to the functions of the interface code package. With this mechanism, the custom package "with's" only one package, the interface code package, the specifications of which can be made available to the student.

The custom package (both specifications and body) in its entirety is made available to students. Since the custom package derives all its lower level functionality from only one package, it is more easily comprehensible. This makes it easier to maintain and update. Also, the calls from the application package are at a much higher level. This makes the use of the various functions of the custom package from the application package easier. Figure 2 shows the construction of packages using our approach.

An obvious question that one may raise is: Why do you need the custom package? Why not have its functionality available in the interface code package itself? The answer to that is: *ease of use*. If the higher level functions are made available in the interface code package, the package becomes more complex. To make the student understand all the code would become very difficult. Remember, we are dealing with freshman and sophomore level students. By

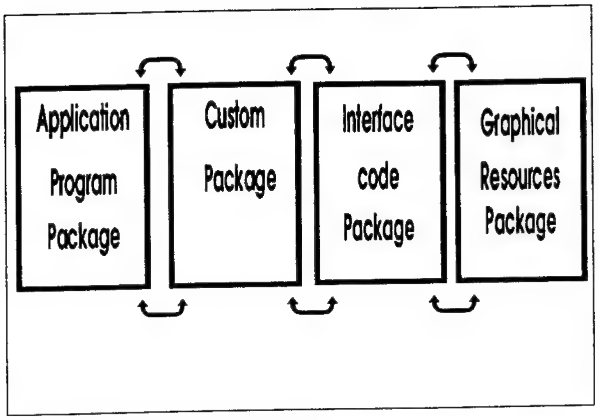


Figure 2: Package relationships using our approach

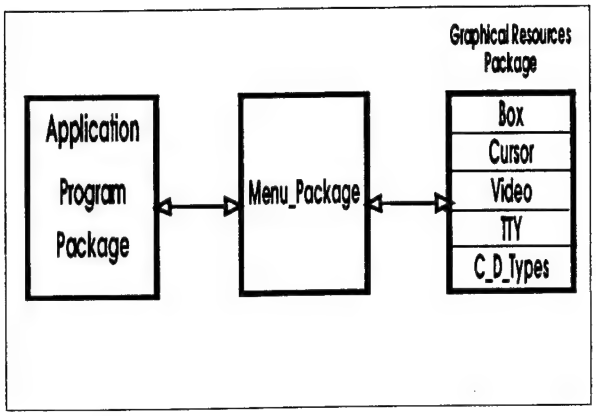


Figure 3: Menu_Package Design using conventional approach

making the custom package available to the students, it gives the students enough power to make variations to suit their application. As the students move to higher level courses, the lower level details of the interface code package could be made available, enabling them to write their own custom packages.

We have used the Meridian Ada compiler (for DOS and for Microsoft (MS) Windows 3.1 environment) for generating graphical elements such as menus, dialog boxes, windows, and graphics. Below we present two examples illustrating the use of the menu package for creating menu programs in the DOS and MS Windows environments.

Example 1: The use of a DOS menu program:

The Meridian DOS compiler provides packages such as Box, Cursor, Video, Tty, etc, for controlling aspects of the video and drawing graphical elements. Using the conventional approach, the design is as shown in Figure 3.

The Menu_package is the interface code package. It uses ("with's") packages such as Video, Cursor, Box, Tty, and Common_display_types provided by the system to handle lower level graphical elements. These lower level packages constitute the graphical resources. The application package calls the Menu_package. Examples of functions, provided by the application package, to control the menu would be: turn the borders on or off, turn colors on or off, and set the menu position on the screen. To have a higher level of functionality, such as setting menu colors, border types, etc., the application program would have to set parameters to function calls in the menu package. The packages and their contents are shown in Figure 4.

Our design of the menu package uses a custom package, called My_menu, in between the Menu_package and the application program, as shown in Figure 5.

Specifications of Menu_Package

```
With Cursor;
With Video;
With Box;
With TTy;
With Common_Display_types;
Use Common_Display_types;

PACKAGE Menu_Package IS

...

TYPE Arr_type IS ARRAY ...          -- Holds info of choices
TYPE Menu_type IS RECORD
   Row   : Integer;
   Col   : Integer;
   Options : Integer;
   Colors : Boolean;
   Borders : Boolean;
END RECORD;

PROCEDURE Display_menu (Options : Menu_type;
                        Choice  : OUT Integer);
PROCEDURE Clear_screen;
PROCEDURE Set_color (BK_Color : Color_type;
                     FG_color : Color_type);
PROCEDURE Set_Border (Border : Border_type);
```

Body of Menu_Package

```
PACKAGE BODY Menu_Package IS

PROCEDURE Clear_Screen IS
  BEGIN
     Video.Clear_screen;
  END Clear_screen;

PROCEDURE Display_menu (Options : Menu_type;
                        Choice  : Out Integer) IS
BEGIN

Cursor.Move (...)                 -- Calls to functions of graphics packages
   ...
   ...
Box.Draw (...)
   ...
END Display_menu;

PROCEDURE Set_color (BK_Color : Color_type;
                     FG_color : Color_type) IS
BEGIN
   ...
  Display_attribute (Foreground => ...,
                     Background +> ...);

END Set_color;

PROCEDURE Set_Border (Border : Border_type);
...
```

Figure 4: Details of Menu_package using conventional approach

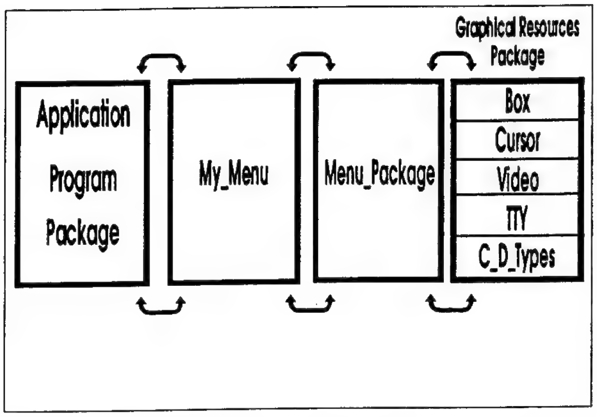


Figure 5: Menu_Package Design using our approach

This package "with's" the Menu_package and allows higher level functions, in addition to the ones provided by Menu_package. These higher level functions include Set_border_dblline (which sets the border to double lines), Set_border_colors, Set_highlite_color, etc. The operations of these functions are achieved by using appropriate calls to the functions provided by the Menu_package. Hence the Menu_package of our approach provides more functions than the one in the conventional approach. The skeletal structure of these packages is shown in Figure 6.

Example 2: The use of a menu program using Windows:

MS-Windows is a message-passing, event-driven operating system. Integrating it with a procedure-oriented language such as Ada requires a proper understanding of the interaction of the runtime system with the main program and the associated libraries which provide the GUI services. The GUI libraries of Windows are quite large and the functions provided are too many for a beginning student to comprehend. Writing even small programs which use limited capabilities could be a major task because one would not know which library or function to use.

Using our approach, we have created a custom package called My_win_menu which provides custom menu-related functions for Windows to the application. This package hides many of the details of Windows concepts, making it easier to use.

Work in progress

Windows based packages which have the correct level of abstraction are more difficult to construct. We are in the process of developing a few more packages which can be illustrated early on.

Ideas for other courses:

Some of the problems which can be demonstrated using simple graphical interfaces are:

CS1 Level:

In addition to the use of menus, other interesting concepts which could be used are the drawing of other elementary graphical elements such as lines and boxes to make various shapes, demonstration of single and multi dimensional array manipulation.

CS2 Level:

Problems at this level could include the demonstration of path finding using a maze.

Algorithms Level:

Animation of graph traversal could be a good application.

Specifications of My_Menu

```
With Menu_Package;

PACKAGE My_Menu IS

...

TYPE Arr_type IS ARRAY ...           -- Holds info of choices
TYPE Menu_type IS RECORD
   Row   : Integer;
   Col   : Integer;
   Options : Integer;
   Colors : Boolean;
   Borders : Boolean;
END RECORD;

PROCEDURE Display_menu (Options : Menu_type;
                        Choice  : OUT Integer);
PROCEDURE Clear_screen;
PROCEDURE Set_color_bk_yellow;
PROCEDURE Set_color_fg_green;
PROCEDURE Set_Border_singleline;
PROCEDURE Set_Border_doubleline;
...
...
```

Body of My_Menu

```
PACKAGE BODY My_Menu IS

PROCEDURE Clear_Screen IS
  BEGIN
     Menu_Package.Clear_screen;
  END Clear_screen;

PROCEDURE Display_menu (Options : Menu_type;
                        Choice  : Out Integer) IS
BEGIN

   Menu_package.Move (...)               -- All calls are to functions
      ...                                -- in Menu_package
      ...
   Menu_package.Draw (...)
   ...
END Display_menu;

PROCEDURE Set_color_bk_yellow IS

BEGIN
   ...
  Menu_package.Set_color_bk (Color => Yellow);

END Set_color_bk_yellow;


PROCEDURE Set_Border_single_line IS
```

Figure 6: Details of My_Menu using our approach

Conclusion

We believe that the challenge is up to the instructors to make Ada interesting and exciting for undergraduate students. In this paper, we have outlined our approach of how it can be done. We have demonstrated our approach using some examples. We are in the process of developing more examples based on our approach. The tools (such as Meridian Ada compiler for Windows, Integr Ada compiler for Windows), though not very robust, are available to program some eye-catching graphics systems. We as instructors have a challenge to develop and package this code, such that it can be easily used for building bigger and better modules.

TEACHING THE CONCEPT OF REUSABLE SOFTWARE COMPONENTS WITH ADA

Abdullah Al-Dhelaan and Jagdish C. Agrawal
Computer Science Department, King Saud University

Abstract

The effort required to specify, design, code, and test a program unit takes much longer time and effort than understanding the specification of a previously used, well-designed, and tested program unit for the same requirements. It may be hard though to catalog program units for each set of requirements. This is where the abstraction and information hiding principles of software engineering and the power of Ada to handle generic program units provides an important base for teaching the principle of reusability.

This paper provides an overview of the techniques used, and enforced for teaching the concept of reusability and making the students practice these concepts. The success of the method was measured by how well the students were able to write, read, and interpret correctly, the specifications of program units, and were able to use previously built components, rather than remake similar parts. This was accomplished through the questions on the examinations that emphasized this. This paper reports the preliminary findings on this.

Students as well as practitioners of Software Engineering need tools that support software reusability. Above all, their tools must comply with ANSI or ISO standard to ensure easy portability and interoperability of the reusable components. Ada exhibits strength in providing some support for reusable components. However, we felt that an on-line search tool that could search the contents of a library (to be dynamically updated, as more parts are added to the library) and provide relevant information on the queries made, similar to the Windows help facility, would have been useful. Our students did not have such a tool, and while there were some proposals to build a Windows based search tool, there wasn't enough time to do that. We hope that Ada 9X designers will consider this suggestion.

1.0 Introduction

In the four year Computer Science Curriculum at most places, there is a lot of emphasis on building software. We also need to teach these students the concept of reusability like in mathematics, when a class of problems is solved, the next class of problems can use that solution without having to "reinvent" the first solution. In mathematics, we specify already solved problems in theorems and lemmas. Ada also provides the mechanism of ***package specification*** for clearly specifying the solutions contained in a package body. With a good and properly indexed catalog consisting of package specifications, programmers could select various packages just like spare parts for an automobile.

The Ada programming language provides a rich capability for construction of user defined data types at varying levels of abstraction. Ada's encapsulation mechanism in the form ***of package*** construct offers strong help in integrating reusability of components into proven methods for the design of new software systems and repair of existing software systems. Ada's package specification mechanism assists in defining the external interface of software components.

Chase and Gerhardt[1] as well as Booch[2] have advocated the use of Ada for component design even for systems that are to be built using other languages. Naturally, Ada can be used to specify the ADT's we wish to teach in Data Structures classes in the four year Computer Science Curriculum. Of course, Ada offers everything that Pascal does and much more for teaching a Data Structures course. The fact that the packages can be compiled, carefully tested and put in a library[3], makes the ADT's built by the students reusable. With this philosophy, we introduced the concept of reusability in the Structured Programming course to sophomores. The syllabus requirements in that course mandated use of Pascal. The students were given exercises in clearly specifying all the components they made and preparing a user's manual for these. In a senior level course, Software Engineering II, Ada's generic program units were introduced to support our lessons on reusability. This paper provides some overviews of such efforts.

Another area that needs to be researched, developed and integrated in the Computer Science curriculum is testing of reusable components. Ada packages offering reusable special purpose ADT's are fine and wonderful, but their testing needs some research. By including reusability of ADT's and their testing in a data structures course, we will bring the Data Structures course forward from the 70's to the 90's. This paper shares the experiences of the authors in this matter.

2.0 Reusability with ANSI Pascal

Before the beginning of the class, we devoted a lot of time investigating what program units in ANSI Pascal could be used as examples to the students to not only illustrate reusability, but also require them to practice reusability. We had to find applications that would require common resources and services not offered by ANSI Pascal. Dynamic stacks and queues built with the help of pointers, and static stacks and queues built with the help of arrays were good examples, but each time the element type in the stack or queue changed, the code of the component will have to be modified, recompiled, and retested. We felt that such examples should be reserved for the senior level class, where we would be able to use Ada.

While looking for a readily reusable component, we found that all students had the need for doing text input/output to records containing variable length strings, and modifying them. Although, ANSI Pascal does provide the predefined types real, integer, char, Boolean, and text (text file), it provided no variable length string type. Therefore having them build a type VarString could be used to illustrate the use of reusability. We found that the students found the exercise very satisfying and gave them a sense of accomplishment.

Our students were also exposed to complex numbers in their mathematics courses, and we felt that a reusable program unit for type complex would be very useful. Students were asked to write the requirements for it, and then we went through a refinement process to finally write the specifications for each function or procedure needed. Subsequently the students built these components, tested them and reused them with different applications. The best specifications of reusable components built by the students are given in the next section.

3.0 Two Reusable Pascal Components

After the students went through the exercise of specifying the two reusable components described in the previous section, the second author, who was the instructor, wrote the specifications for the two components, deriving the best ideas from all implementations, and distributed these to the class for reuse. The instructor had put the implementation in the library already. Basically, the specifications became like a catalog. These specifications are described below for the readers who may want to use a similar exercise:

3.1 VarString component

ADT VarString was built with the structure of a variable length string and a minimum set of functionality for it. First we describe the structure, and then in section 3.1.1, we will describe the specification of the functionality. Principles of software engineering require the implementation details of the reusable parts hidden from the re-users. Most important challenge in making software components reusable is giving careful and detailed description of the specifications. That is our goal in this example and the one described in section 3.2.

```
const
   Max = 256
         {If VarStrings of larger capacity are
         needed, the constant value will have to
         be changed}
type
   VarString = packed array [1..Max] of Char;
         {In the array, the presence of a null
         character in an element's value would
         indicate the end of the string. Thus, for
         I/O, one would most likely use a top
         tested loop, continuing the loop while
         the ith character is not null.}
```

3.1.1 Functionality for VarString

```
function Length (S: VarString) : Integer;
         {This function counts the number of
         character in S until the first null
         character is encountered and returns
         that value.}

procedure ReadVarString (var F:text;
               var S:VarString);
         {This function reads from the text file
         F one character at a time and inserts it
         in S[i], until the end of file, at which
         time it inserts a null character in the
         next S[i] and returns the VarString S.}

procedure WriteVarString (var F: text;
               S:VarString);
         {This function writes one character at a
         time from  S[i], in the text file F until a
         null character is encountered in S[i].
         The null character is not copied to F,
         instead, F is closed and control
         returned to the calling program unit.}

function GetChar (S:VarString; N:Integer):Char;
         {It returns Nth character in the
         VarString S.}

procedure PutChar (var S:VarString;
               N: Integer; Ch:Char);
         {It inserts the character Ch in the Nth
         position in the VarString S.}

procedure Retrieve (Source: VarString;
               N,K:Integer;
               var Target:VarString);
         {It retrieves a substring of Source,
         starting at the Nth location of size K,
         stores it in Target and returns it to the
         calling program unit.}

procedure Combine (Source1:VarString;
               Source2: VarString;
               var Target:VarString);
         {It puts the Source2 at the tail-end of
         Source1, and puts the so concatenated
         string in Target. If the size of the
         concatenated string is bigger than Max,
          the excess is thrown away.  Target is
          returned to the calling program unit.}

function Position (Sub:VarString;
                Main:VarString) : Integer;
          {It looks if Sub is a substring of Main,
          and if so, the function returns the
          location where Sub first starts to show
          up in Main.  If not, the function returns
           zero.}

procedure Delete (var Given:VarString;
                N, K:Integer);
          {It deletes K characters, beginning at
          location N in the Given VarString, and
          returns the changed value of Given.}

procedure Insert (var Given:VarString;
                N :Integer; Sub:VarString);
{It inserts the substring Sub, beginning at
location N in the Given VarString, and returns
the changed value of Given.}
```

3.2 ComplexNum component

ADT ComplexNum was built with the structure of a record containing its real and imaginary parts, x and y respectively, and a minimum set of functionality for it. First we describe the structure, and then in section 3.2.1, we will describe the specification of the functionality.

```
type
   ComplexNum = record
                x : real; {real-part}
                y : real  {imaginary part}
          end; {ComplexNum}
```

3.2.1 Functionality for ComplexNum

Much like what is described in section 3.1.1, the functionality for the ComplexNum ADT was specified to provide following capabilities:

- Initialize a ComplexNum with value (0.0, 0.0)
- Create ComplexNum from two given real numbers x and y.
- Input a complex number in an interactive manner.
- Output a complex number enclosed in parentheses, as an ordered pair of real part and imaginary part separated by a comma.
- Add two complex numbers and return the sum.
- Subtract ComplexNum Z2 from ComplexNum Z1, and return the value of (Z1-Z2).
- Multiply ComplexNum Z1 and Z2, and return the value of their product.
- Divide ComplexNum Z1 by Z2, and return the value of their quotient.
- Get RealPart of ComplexNum Z.
- Get ImagineryPart of ComplexNum Z.
- Return absolute value of a ComplexNum Z.

The reader may request directly from the authors, complete specifications in Pascal. We are leaving those out to save space.

4.0 Reuse of Ada's Generic Program Units

The templates provided by Ada's generic packages and subprograms provide excellent tool for creating reusable components. Users of such reusable components provided by these generic templates simply create a specific instance of the generic unit by making a local copy through a process called instantiation by providing an identifier that names the regular program unit and maps generic parameters with actual parameters similar to the matching between actual and formal parameters in Pascal.

For our senior level students in Software Engineering II, we emphasized increasing reusability by building generic objects. For example, the students were asked to build generic Stack object that is far more reusable, than a Stack object containing specific type of elements in the stack: The students used a template of the type:

```
generic
   type type_of_object is private;
   max_size : natural := 5000;
package generic_stack is
   stack_full : exception;
   stack_empty: exception;
   procedure push (object : type_of_object);
   function size_of_stack return natural;
   function copy_top return object_type;
   procedure pop;
end generic_stack;
```

Every time the students needed the Stack ADT, they instantiated the generic_stack to create specific_stack with a statement like:

```
package specific_stack is new generic_stack (
          type_of_object => specific_type,
         max_size => 100);
```

In the example above, the essence of stack was created with a very high level of abstraction, and hiding the type of objects being stacked and the maximum size.

5.0 Higher Quality with Reusable Components

The software components marked for reuse mature over time with reuse. While the initial users of these components take a higher risk, they also become most familiar with the process of reusing. In the long run, the reusable components become well tested and reliable. The users benefit from this quality and their over all cost for quality assurance for systems built with these components becomes less, than if they had built the code for the functionality for which the reusable component was used.

Ultimately, software engineering will reach a level that the catalogs of the libraries of reusable software components will become as common and reliable as the catalogs of VLSI chips to hardware designers. But to reach that level, we, the computer science educators will have to accept the importance of standards, and also start teaching software reusability. We hope that the ideas presented in this paper will stimulate further discussion and action.

Reuse of existing packages can also greatly increase productivity of software developers. Software reuse has the potential of becoming the greatest productivity enhancer. But the universities need to play an important role in this very important step forward. The biggest challenge in reusability is identifying the requirements for what can become reusable components. The examples given in this paper are primitive and elementary, but they do give some idea of the direction in which we need to move. University educators can contribute a lot by proposing requirements for specific reusable components, in forums like this. Perhaps, some authors will write books with collections of proposed reusable components.

6.0 Conclusions

Ada and ANSI Pascal can both be used for teaching how to build reusable software components, and how to reuse available components. However, Ada has stronger facilities for these concepts, and for this reason, it is important that we, the computer science educators, seriously consider adopting Ada for the computer science curriculum in the core courses, where the early programming habits of the young budding computer scientist are formed. It is true that many nonstandard versions of Pascal, like Turbo Pascal, do offer better facilities than ANSI Pascal. The success, popularity, and proliferation of nonstandard versions of Pascal that claim to offer some of the Ada like features further confirms our conclusion that our students will be better served with Ada. Ada has been around for over a decade now, and there are numerous good text books and good Ada compilers, universities will be doing their software engineering students a big favor by adopting Ada in their programming course sequence. However, we also need to find ways of giving education on software reuse to our students.

7.0 References

1. A. I. Chase and M. S. Gerhardt, "The case for full Ada as a PDL," Ada Letters, ACM Ada TEC, November 1982.

2. G. Booch, "Describing Software Design with Ada," SIGPLAN Notices, ACM SIGPLAN, September 1981.

3. J. Ichbiah, J. Barnes, and R. Firth, Ada Programming Course, La Cella Saint Cloud, France: ALSYS, 1980.

8.0 Authors' Biographical Sketches

Name: Abdullah Al-Dhelaan
Address: P. O. Box 51178, Riyadh-11543
Saudi Arabia
Email: F60C018@SAKSU00.BITNET

Dr. Abdullah Al-Dhelaan is currently the Vice Dean of Admission and Registration and Assistant Professor of Computer Science at King Saud University, Riyadh. Prior to assuming the duties of Vice Dean, he was the Chairman of the Computer Science Department at King Saud University. Dr. Al-Dhelaan has also served as the Director of National Information Center, Ministry of the Interior., Kingdom of Saudi Arabia. Dr. Abdullah Al-Dhelaan is on the faculty of the College of Computer & Information Science, King Saud University, since early 1989. His areas of research are Software Engineering, Parallel Processing, Distributed Systems, and Interconnection Networks.

Name: Jagdish C. Agrawal
Address: P. O. Box 51178, Riyadh-11543
Saudi Arabia
Email F60C029@SAKSU00.BITNET

Dr. Jagdish C. Agrawal is Professor of Computer Science at King Saud University, Riyadh, Saudi Arabia, since January 1990. He has served as Department Chair at Purdue University, Ft. Wayne (1981-82), Division Chief at the Army Institute for Research in Management, Information & Computer Sciences (AIRMICS), Department of Defense (1983-86), and as Computer Science Department Chair at Embry Riddle Aeronautical University, Daytona Beach, FL (1986-90). He also served as Professor of Mathematics and Computer Science at California University of PA (1969-86) from where he took early retirement to become Professor Emeritus. The software engineering projects that Dr. Agrawal worked on at AIRMICS included ARS, STEP, SRT, Follow on Evaluation of PATRIOT etc. He has over seventy publications.

Cognitive Analysis, Design and Programming: Toward Generalization of the Object Oriented Paradigm and the Reuse and Conversion of Legacy Code into Ada

Joseph M. Scandura, Ph. D.
University of Pennsylvania
Intelligent Micro Systems
jms@pobox.upenn.edu

In my talk, I will introduce the rudiments of a cognitive approach to software analysis, design and programming. I will also touch on two additional issues: a) The relationship of cognitive programming to object oriented programming as in C++ and Ada 9X and b) The reuse and conversion of legacy code into Ada which the cognitive paradigm supports.

Cognitive Analysis and Design

This cognitive approach has been motivated by earlier work in analyzing cognitive processes, specifically by a method I called Structural Analysis (Scandura, 1971, 1977, 1982, 1984). (NOTE: The "al" at the end is intended since the work was quite independent of Wirtz's contemporaneous structured programming and structured analysis and design which came later.) The essence of the Cognitive Approach involves modeling system behavior successively from the highest levels of abstraction until contact is made with available data and computational resources. Barriers between analysis, design and programming are effectively eliminated. All systems involve input, output (which may be the same) and operations which map the first onto the second.

At the level of analysis, all systems modeling begins by labeling the abstract input, output and operation. In modeling an event driven system, for example, we initially refer to the incoming events collectively by assigning them a generic but descriptive label. For example, we might refer to "PRODOC (key presses and mouse) events" where PRODOC is the name of a complex software engineering application. Similarly, the initial output refers to the collective outputs of the application -- say, "software engineering outputs". The top level operation, then, might be described as "software engineering operations".

Clearly, at this level of abstraction, we do not know very much about the application other than its domain of applicability, and then only in a very generic sense. Notice, however, that inputs (i.e., "PRODOC events") and outputs correspond to abstract objects, much like a base class in C++ or a top level package in Ada 9X. As we shall see, we will find operations lurking inside as we probe more deeply. The abstract operation "software engineering operations", however, is different. It acts on and produces objects as wholes, not by referring to specific procedures or functions contained therein (or equivalently by passing "messages" to the objects).

Paralleling structured analysis, this abstract triple (domain, range and operation) can be refined into a sequence, a selection (e.g., IF ... THEN) or iteration (e.g., LOOP). One can also envisage parallel execution. Unlike structured analysis, however, we must also refine the abstract data. If the refinement is a sequence, then the abstract input and output remain as before. If the refinement is a selection, however, we must introduce an "extra domain" condition variable or relation which partitions the domain. Domain elements in one element of the partition serve as input to the THEN statement. Inputs that fall in the other element of the partition are input to the ELSE statement. Clearly, selections can be generalized to CASE structures involving any number of elements in the partition. Since the output of the original abstract operation must reside in the abstract range variable it follows that the output of each CASE alternative must also lie in that same range variable.

The case of iteration can be viewed as a variation on binary selection in which the THEN statement, or LOOP body, accepts inputs in both partition elements. Outputs of the LOOP body statement or operation must then be contained in the abstract domain variable with elements in both partition elements. The hard part is to insure that the LOOP body statement or operation modifies the outputs in a systematic way so as to insure a transition between the two partition elements of the domain (one of which exits the loop).

One might also refine an operation into two or more operations which act in parallel on the domain and collectively produce elements in the range.

The above analysis deals essentially with internal processes. This leaves the question of how to deal with inputs and outputs. Abstract inputs clearly do not provide all of the detail necessary for reflecting the complexity or structure of the incoming data. In this context, for example, partition elements of the partitioning class of a selection effectively introduces new domain variables, in particular subsets of the original abstract domain. The abstract domain, then can be computed by defining a function (i.e., taking the union of) on the partition elements. This is perhaps a classic case of "evaluation". Similarly, abstract output must be made more concrete to reflect realistic output structures. This is accomplished by "inheritance" where concrete elements effectively "flesh out" more abstract ones.

Notice that the above analysis has much in common with traditional "object oriented analysis". We have introduced new lower level objects which share features in common with the "base" object. We have also shown how both evaluation and inheritance flows naturally from the analysis.

In my talk I shall elaborate on the above. In particular, I would like to call attention to a natural progression inherent in the process. Abstract operations and objects can be refined successively and indefinitely. There is no natural or required transition between analysis and design. Indeed, as I note below, there is no sharp distinction between design and programming, other than perhaps shifting from a generic representation to a specific one (e.g., to a programming language like C++ or Ada 9X). Successive phases of refinement flow naturally from one to the other. The only divisions which exist will be arbitrary and strictly for convenience.

The fact that data (objects) and process (operations) are always represented at the same compatible level of abstraction has important implications for testing -- in two words "design debugging".

Abstract domains and ranges contain abstract objects. These objects are operated on by the corresponding abstract operations. Consequently we can, in principle, imagine executing such models at arbitrary levels of abstraction. (Parenthetically, the PRODOC system referred to above executes such specifications making it possible to simulate and debug systems as they are being developed.) As I have shown elsewhere (Scandura, 1990, 1992), the ability to debug design from the highest levels of abstraction dramatically reduces the number of empirical tests required to "prove" a system. When testing is delayed until after a system has been implemented, the number of paths to be tested goes up exponentially with complexity. This number only goes up additively when testing is done at successive levels from the highest levels of abstraction.

Relationship Between Cognitive Programming and Object Oriented Programming

In his defining book on C++, Stroustrup (1991) makes a useful distinction between enabling a paradigm and supporting one. A language supports a programming style if language facilities make it convenient, "reasonably easy, safe and efficient" to use that style. If it takes exceptional effort or skill to write programs in a given style, then that language merely "enables" the technique. As an example Stroustrup observes that one can write structured programs in Fortran and object oriented programs in C, but it is difficult to do so because these languages do not directly support those paradigms.

Stroustrup's consideration of programming styles and key language mechanisms for supporting them is also informative. *Procedural programming* was the original and probably still the most commonly used programming paradigm: Decide which procedures are needed and use the best algorithms available. In *modular programming* emphasis shifts from the design of procedures toward the organization of data. Here one decides which modules are needed and the program is partitioned so that the data is hidden in the modules. *Data abstraction* extends this idea by centralizing data according to type. The basic paradigm involves deciding which types of data are needed and providing full sets of operations for each such type.

In introducing *object oriented programming,* Stroustrup denotes a major limitation of abstract data types -- namely the inability to adapt abstract data types to accommodate new uses without explicitly modifying their definitions. Object oriented programming, of course, extends the idea of abstract data types by introducing inheritance. In this case new classes are formed from existing ones via specialization. It should be noted incidentally that more abstract types also might be derived from simpler ones.

In object oriented programming, notice that we do not have any operations on objects per se. Rather, operations outside of objects consist essentially of calls (or messages) to functions *within* objects. Reference to both the objects (e.g., Ada packages) and the internal functions or procedures is necessary.

Cognitive programming extends this paradigm by allowing operations on objects as a whole, without reference to the internals, functions or otherwise. Consequently, cognitive programming can greatly simplify programming representations. It also generalizes the notion of object orientation. Not only do objects map directly into the real world but so do the abstract operations on those objects. One can describe systems in ways which directly maps into the way people think about what needs to be done -- whether these abstract entities refer to nouns (i.e., objects) or verbs (i.e., actions).

It is constructive in this context to consider the distinction between operations which act on objects as a whole and operations (e.g., functions) defined within the objects. These two types of functions or operations have direct parallels in human cognitive processing. Functions defined within objects correspond rather directly to automated perceptual processes, things which take place more or less automatically without conscious thought. Operations on objects as a whole refer to more conscious cognitive processing. Although this may all seem even more abstract than object oriented programming most people find cognitive programming much more natural. I shall give examples in my talk.

Some examples from new capabilities in Ada 9X are: programming by extension, class wide programming, hierarchical libraries and generic parameters.

Reuse of Legacy Code

The above cognitive paradigm, especially when coupled with design debugging provides a natural foundation for the reuse of legacy code. Abstract cognitive models of given application domains often map naturally onto components derived directly from legacy code in those domains. These components might be derived via reverse engineering of the simple legacy code and/or they might be derived via reverse engineering and/or conversion of that legacy code into Ada (Scandura, 1994). While not stressed in the above, an incidental benefit of cognitive programming is that it minimizes if not eliminates the need for global variables. Since parameters (e.g., domain elements) may be as abstract as necessary, one is not forced into using long parameter lists. This has obvious implications for system reliability and maintainability.

To enable reuse, the only essential in legacy code is that the top level legacy components be essentially free of global variables. Calls based largely (if not exclusively) on parameter passing, allows direct, clean mappings between high level designs and the reused components.

Conclusions

By way of conclusion, the above represents only some preliminary thoughts on cognitive analysis, design and programming. It is based, however, on a long history of research in analyzing cognitive processes. Indeed, cognitive programming as I have described it represents only the first of two major steps in generalizing object oriented programming. The second corresponds to the notion of higher order rules or knowledge (e.g., Scandura, 1971, 1973, 1977). Higher order knowledge corresponds effectively to programs that may operate on and/or produce new programs. I look forward to an exciting exchange of ideas with the audience.

JOSEPH M. SCANDURA, PH. D.

Dr. Joseph M. Scandura is an interdisciplinary research scientist in structural learning: the science of cognitive, instructional and intelligent systems engineering. He serves as president of Intelligent Micro Systems and has been instrumental in its receipt of five highly competitive Small Business Innovative Research contracts. He is well versed in the CASE and reengineering markets and has managed the development of several lines of productivity software. These include advanced intelligent authoring systems and culminate in the *PRODOC re/Nu Sys and re/Nu Conversion Workbench,* a full life cycle design, development, maintenance and conversion system. He was recently awarded a US Patent *on Displaying Hierarchical Tree-like Designs in Windows.*

He is a professor at the University of Pennsylvania and author of over 170 scientific publications including eight books. His article "Deterministic Theorizing in Structural Learning: Three Levels of Empiricism" was selected as a Citation Classic by ISI's *Current Contents.* Recent publications include "Automating Renewal and Conversion of Legacy Code: A Cognitive Approach," *IEEE Computer,* 1994, in press. In addition to his research, he is Editor in Chief of the *Journal of Structural Learning* and has organized various international interdisciplinary conferences including a ten day *NATO Advanced Study Institute on StructuraLlProcess Theories.*

Dr. Scandura received his B.A. and M.A. degrees from the University of Michigan and his Ph.D. degree from Syracuse University. He also taught at the State University of New York, Buffalo, Syracuse University and Florida State University. He has been a postdoctoral research follow or scholar-in-residence in experimental, mathematical and educational psychology and artificial intelligence at Indiana University, University of Michigan, ETS, Stanford University and MIT, respectively. He completed a senior postdoctoral fellowship for one year at the Institute for Mathematical Studies in the Social Sciences at Stanford University and one-half year at the IPN Research Institute in Kiel, Germany. A Fellow of the American Psychological Association, Dr. Scandura won Fulbright Awards to West Germany in 1975 and 1976. He appears in the 43rd Who's Who in America and the International Who's Who in psychological science.

Contact at:

Dr. Joseph M. Scandura
Scandura (div. of Intelligent Micro Systems)
822 Montgomery Ave., Suite 317
Narberth, PA 19072-1937
215-664-1207
jms@pobox.upenn.edu

EXPERIENCE IN TESTING OBJECT-ORIENTED ADA SOFTWARE

Amy P. Graziano
IIT Research Institute
185 Admiral Cochrane Dr.
Annapolis, MD 21401

SUMMARY

This paper describes the lessons learned from testing Ada software developed during a recently completed project that used object-oriented analysis and design methods. The project required that the new software be developed as a library of reusable components that provide the same functionality as existing structurally-designed FORTRAN software. This paper covers the project background, testing process, lessons learned, process improvement initiatives, and conclusions.

BACKGROUND

IIT Research Institute (IITRI), under a contract with the Department of Defense (DoD), developed automated tactical planning tools for the United States Army. In 1991, our organization was tasked with developing scientific application software for a large Army command and control system. The primary baseline for the development effort was an existing software system that was written in FORTRAN and C; and was designed using functional decomposition. The new application software was developed by re-engineering the existing baseline algorithms, using object-oriented analysis and design techniques and implementing the new design in Ada. The new software was delivered as a library of reusable objects and a library of functional drivers, which integrate the objects to perform specified functional requirements.

Ada packages were used to implement each object in the delivered library. In general, there was a one-to-one relationship between an object and an Ada package in the application software. Combinations of packages and stand-alone procedures were used to implement functional drivers.

The software development team was organized into the following groups under a project manager: Quality Assurance, Configuration Management, Software Engineering, and Testing. The Software Engineering Group was responsible for the analysis, design, implementation, and initial high-level testing of the software and its components. The Quality Assurance Group was responsible for establishing and enforcing quality, style, and process guidelines. The Configuration Management Group was responsible for version control of all baseline and development software. The Testing Group consisted of individuals with varying expertise and backgrounds: electrical engineering with testing and object-oriented training, domain expertise with actual field experience, and software engineering with testing and object-oriented training. Testing Group personnel were responsible for detailed independent unit and integration level testing for all deliverable products. The Testing Group was established at the start of the third year of the three year development phase.

TESTING PROCESS

The Testing Group was responsible for testing all objects and functional drivers prior to a formal acceptance test by the government. The Testing Group's primary goal was exposing errors in analysis, design, and code products as early as possible. Along with members of the Software Engineering and Quality Assurance Groups, testers attended analysis, design, and code reviews with an eye on finding errors and missing requirements. In addition to these static reviews, the Testing Group dynamically executed tests on the code in an attempt to detect errors.

Testing Group personnel developed unit test drivers that enabled each object to be tested as a single executable unit. Developing these drivers involved writing standardized Ada code that allows a tester to individually execute each visible operation in the object's specification. In addition to test drivers, the Testing Group was responsible for developing formal test case descriptions, test input data, and expected and actual test results for both objects and functional drivers. In general, expected results were those obtained by running the baseline software, but in some cases were adjusted to accommodate any necessary changes to the baseline functionality as development progressed.

Black-box test cases are based solely on the specifications for a particular unit to be tested. White-box test cases, on the other hand, are based solely on the source code for the software.
Black-box and white-box test case descriptions were developed for each object. Functional driver test case descriptions were generated based on a black-box approach. Functional driver tests focused on finding errors in specified functional requirements. Black-box tests were developed by examining the object or functional driver specification and writing tests based on the advertised functionality. White-box tests were generated by an automated tool that analyzed each object's implementation (Ada body) and produced path descriptions for each path to be tested. All test descriptions included expected results. Expected results for white-box test cases were not generated by the automated tool and were, therefore, generated manually. Test case descriptions were formatted

and managed using a word processor.

Following the development of object test drivers and test case descriptions, object level testing was performed. Testers executed the drivers and entered the data required to perform each individual test case specified in the test case descriptions. Test drivers captured all input and output data and saved the data to files for comparison during regression testing.

System level testing was performed for each deliverable functional driver. Functional driver testing required developing test input data which met the specified requirements of each test. The functional drivers were executed once for each test case. Again, input and output data were saved for comparison during regression testing.

Objects and functional drivers were regression tested as needed. Each time the code that an object or functional driver depended on was changed during development, regression testing for that object or functional driver was performed. The input files created during the initial test were modified, if necessary, and then used to re-execute the tests. Files containing the expected output were also modified, if necessary, and were compared against the new results. The Make utility, provided with the Ada compiler, determined if an object or functional driver needed to be re-compiled or its executable re-linked and, therefore, re-tested. Shell scripts were used to facilitate automated regression testing on a periodic and as-needed basis.

Deviations between expected and actual test results required further investigation. Test personnel determined if the deviation was caused by an incorrect test case, a problem with the test driver code, or a problem with the object or functional driver code. Incorrect test cases were investigated and corrective actions taken. Problems with test driver code were corrected by Testing Group personnel. Problems with objects or functional drivers were turned over to a designated software engineer for correction. Tests were re-run after problems were fixed.

LESSONS LEARNED

Although IITRI-Annapolis has been developing Ada software since 1986, this was the first object-oriented development effort undertaken at our organization. Through its efforts, the Testing Group learned many new lessons about planning and executing software testing efforts. The predominate challenges associated with testing this software are described below.

The testing process should be integrated into the entire development life-cycle. The software was developed following an incremental and iterative life-cycle. At the highest level of abstraction, objects were identified and designed. Complex objects were decomposed into additional levels of abstraction until the complexity of a single object was manageable enough to implement and maintain. During this process, some objects were in implementation stages, while others were waiting for lower level objects to be designed and implemented. Because of the constant changing of objects in all levels of the object hierarchy, the incremental and iterative life-cycle necessitates parallel testing at all levels of the hierarchy. Testers must be involved in the entire development process from the early stages of analysis and design through the final stages of implementation and testing. Testers must be aware of object relationships. They must also be aware of potential changes to existing designs and implementations so they can be prepared to modify test cases, drivers, inputs, and expected outputs. Because of this, Testing Group personnel must constantly communicate with the Software Engineering, Quality Assurance, and Configuration Management Groups.

Regression testing is required throughout the entire testing process. One consequence of an incremental and iterative life-cycle is that low level objects may change in a late iteration of the development process. A change to such an object requires retesting of all higher level objects that depend on it. Due to the potential frequency of such changes, automated regression testing is virtually mandatory in this type of development environment.

Objects must be tested as units. Testing software designed using object-oriented technology is different than testing traditionally developed software. Traditionally, a function or procedure was the unit addressed in unit level software testing. That function or procedure was tested as a stand-alone entity. With object-oriented software, however, the unit, which is addressed, is an entire object (Ada package in the case of the application software). Through the use of private types, Ada enforces the object-oriented concept of encapsulation or information hiding. The structures of private types are not visible to clients of the objects. Visible operations (functions or procedures) defined for a private type are the only operations a client may use to operate on data of that type. Because Ada objects are implemented using private types, the collection of data and operations that constitute the object must be tested together as a single unit. One operation in an object may need to be used to verify correctness of another operation in the same object. For example, to test a Set operation that sets a value, a tester must invoke the Set operation and then invoke the corresponding Retrieve operation in order to verify that the value was correctly set. Prior to this test, the tester must be confident that the Retrieve operation produces correct results. Without the Retrieve operation, a tester would have no method of verifying correctness of the Set operation. Testers must remember that an object, with all of its data and operations, is a single unit. The data and set of operations in an object are intimately related: private data is accessed only through an object's visible operations.

Testers must devise a method of executing white-box tests on fully encapsulated operations.

As indicated earlier, the application software exercises the object-oriented concept of encapsulation, which prevents clients from seeing the implementation of an object. In addition to encapsulation of data, code is hidden from clients in the bodies of Ada packages and operations. This is a desirable practice, since encapsulation allows program changes to be reliably made without side effects to clients. However, it is impossible to perform white-box tests when an operation is completely encapsulated (i.e., has no visible interface) within an object.

White-box testing is important for exercising tests on each path through an operation. White-box tests can be produced for operations that are internal (i.e., implemented in the body with no visible specification) to an object. These tests, however, cannot be executed since the object provides no interface to internal operations. In the case of the application software, many of the most detailed algorithms are implemented in internal operations. The correctness of these operations is crucial to the operation of the object, as well as many functional drivers. To achieve adequate testing coverage, an effective method of white-box testing of internal functions must be devised and applied. Some discussion has suggested that for testing purposes only, a specification be written that exports all operations of the package so that each of them can be independently tested. However, this method contradicts the idea that tests be executed on the identical software that is to be delivered as the finished product.

Testers must define test completion criteria.

Software testers continually debate over the concept of exhaustive versus adequate software testing. Although a one-hundred percent error free system is the ideal of software development, practical limitations often make it impossible to exhaustively test a complex software system. Testers must consider the number of errors detected per unit time, schedule, budget, and operational environment requirements when determining test completion criteria. Coordination with the Quality Assurance Group, the Project Manager, and the customer is required for these decisions. Test completion criteria must be defined prior to commencement of testing.

A strategy for testing generics must be defined.

Ada generics add tremendous complexity to the job of ensuring correctness of a software component. The interface of a generic defines the combinations of generic formal parameters that may be provided by a client. By definition, a generic software component must function according to its specification for any combination of generic formal parameters. For completeness, a generic package or procedure, which has multiple generic formal parameters, must be tested using all permutations of these parameters. This job rapidly becomes extremely time consuming, tedious, and potentially unmanageable. The Testing Group must define a practical approach for adequately testing generics in an Ada system. For a specific application, it may be adequate to test the generic package or operation using only the instantiations required to achieve essential functionality. For other applications, reusable libraries for example, adequate testing may only be accomplished by testing all possible instantiations of a generic component.

Testers and developers must define reasonable limits for reusable software.
Labeling software as "reusable" implies that it can be used over and over again in different environments (hardware/ operating system/compiler/database management system/user interface) and applications. Testers and developers must define reasonable limits for inputs of software that is being developed as reusable. Maximum and minimum values for input parameters must be defined both for testing and reuse purposes. Limits must be selected such that the software component will function according to its specification for all values between the maximum and minimum values. The range of input values specified by the maximum and minimum values must sufficiently cover all required input values for the component. Given these limits, the component must perform as specified in any environment. The reasonable limits must be advertised and enforced by the software.

A strategy for ensuring correctness of software must be formulated when software is re-designed from an existing system.
Testing Group personnel must consider a method of formulating expected results for re-engineered software. Expected results may be gathered directly from the original baseline software or calculated manually. While generating expected results directly from the baseline software is the most straightforward approach, hidden difficulties may arise. Subtle enhancements or bug fixes in the new software can cause difficulties when trying to map expected results to newly generated actual results. The new software may produce different results than the baseline software. Any such differences between the baseline and the new software must be clearly explainable and documented in order for the Testing Group to intelligently decide whether or not expected results should be based on, and generated from, baseline software. If expected results are not to be generated from existing software, they must be manually generated. Depending on the application and its operational environment, expected results may even have to be verified by such means as laboratory testing or field measurement. Manual calculation or independent verification of expected results may consume much of the Testing Group's limited resources. Regardless of which method is used, it is important to make the decision and begin generating expected results early in the testing

process.

Outputs generated from the Test Group must be baselined and configuration managed.
Test process outputs (i.e., test case descriptions, test drivers, test input data, and test output data) must be treated as deliverable products and controlled by some form of configuration management. It is useful to track version numbers of test outputs and map them to version numbers of corresponding software components. At some point after the software is delivered, it may become necessary to retest an older version of the software. Consequently, it may be necessary to recreate the test environment used for the original test of that older version. Test outputs change over time just as the software on which the tests are based changes. During initial test development, version control may seem insignificant, but planning up front to control versions of test outputs will save time in maintenance and in future development phases.

Successful testing efforts require planned resources.
The software was formally tested and accepted by the government in September 1993. Preparation and dedication by the entire project team led to a very successful formal government-witnessed test. Internal independent testing by the Testing Group prior to formal government testing was a crucial step in leading to final acceptance of the system. Project managers must appreciate that the tester's goal is exposing errors in a software system. Managers must plan time and personnel for cycles of error detection, correction, and retesting once software is submitted to the Testing Group. It is imperative that managers ensure that the Testing Group has the resources required to perform adequate software testing.

PROCESS IMPROVEMENT INITIATIVES

The project team continues to maintain and enhance the software under a product improvement program. In addition, development to meet new functional requirements for the system, has begun. In preparation for future government acceptance tests, the Testing Group has continued to perform its original duties and plans to make several changes to its procedures to further increase their effectiveness. These changes are described below.

The Testing Group must work with individual software engineers to aid the engineer responsible for developing an object in creating the unit test driver for that object. This will minimize the amount of coding that the Testing Group will have to perform, enabling testing personnel to focus on detecting errors. Additionally, this approach will ensure that software engineers have a complete and standard unit test driver to use for initial testing. Thus, the software engineers will be required to perform detailed initial testing prior to submitting software components to the Quality Assurance or Testing Groups.

Software engineers will be required to provide sample input and output files from initial testing at the time code is submitted for review. These sample input and output files will be used temporarily for frequent, periodic automated regression testing until the Testing Group produces more formal input and output test files. These preliminary regression test results will notify developers early on if any changes have side effects on other objects. The earlier software errors are detected, the less expensive they are to correct.

Maintenance and version control of test data and documentation is an area that has been targeted for improvement. Examination of methods for maintaining test descriptions and test input data is planned. Currently, the white-box test case generator is run against the new code, and the expected results are entered into the resulting test case document manually. Even though most expected results are the same as in the previous version of the test case document, they must be manually copied into the new test case document. The Testing Group is investigating a more efficient method for updating white-box test documentation each time an object body is changed.

CONCLUSIONS

It is important that a software development project allow for internal independent testing of their products. The tester's focus on identification of analysis, design, and software errors will help to ensure a quality final product. Test planning and resource allocation must be done early in a project life-cycle.

As part of a software engineering process improvement effort at IIT Research Institute, testing experiences are shared. Through experience with software testing, testers will learn methods of improving test processes. As testers continue their efforts, new technical and administrative challenges will constantly be discovered.

ABOUT THE AUTHOR

Amy Graziano is a Software Engineer with IIT Research Institute in Annapolis, MD. She is the Test Group Manager for a large Army command and control system development project. Amy earned a bachelor's degree in Computer Science from Virginia Tech in Blacksburg, VA in 1988. She is currently pursuing a master's degree in Computer Science from Johns Hopkins University in Baltimore, MD.

A SYSTEMATIC OBJECT IDENTIFICATION TECHNIQUE IN THE OBJECT-ORIENTED PARADIGM

Joseph Monroe
Huiming Yu

Department of Computer Science
School of Engineering
North Carolina A&T State University
Greensboro, North Carolina 27411

Summary

Existing object identification techniques are non-systematic and yield non-repeatable results. By non-repeatable, we mean results that vary from one developer to another. In this paper, we propose a framework for identifying objects in a problem domain systematically. This method eliminates much of the subjectiveness associated with object identification by:

a. replacing subjective decision making with repeatable, systematic techniques, and

b. methodically simplifying the problem domain through abstraction and incremental analysis.

Introduction

The identification of objects is a fundamental issue in object-oriented analysis and design. Existing techniques for performing object identification fall into two broad categories: recognition and acceptance; and scenario driven identification.

Recognition and acceptance techniques (1,2,6,7,8) focus on recognizing nouns, or phrases in the problem domain and then testing them for acceptance as domain objects. These techniques typically express what to look for in a candidate object: whole/part structure, roles played , concepts, organizations, known interfaces, etc. The techniques then list acceptance criteria for each candidate. Recognition and acceptance is dependent on an individuals judgment and experience. Candidate object recognition and acceptance testing is open to the subjectiveness of the individual.

Scenario driven object identification uses projection to identify objects in the problem domain. It analyzes the system from the perspective of different users or external entities that interact with the system. Domain functions and function paths are traced through the system to find objects that play roles in the functions. If an object participates in a system function then it is accepted as a domain object. Scenario driven techniques are dependent on an individual perspective of the problem. If two individuals apply the scenario driven approach to the same problem but from different perspectives, they can produce completely different objects. Also the application of scenario driven object identification can often become complicated when system functions branch, and functions paths are not obvious.

Existing object identification techniques are non-systematic and yield non-repeatable results. By non-repeatable, we mean results that vary from individual to individual. We propose **Systematic Object Identification (SOI)**, a method for identifying "relevant" objects in a problem domain in a manner that is systematic. **SOI** eliminates much of the subjectiveness associated with object identification by:

- replacing subjective decision making with repeatable, systematic techniques, and

- methodically simplifying the problem domain through abstraction and incremental analysis.

Systematic Object Identification

Many techniques from scenario-driven object identification are incorporated into SOI:

This work is supported by CECOM Center for Software Engineering, Department of the Army, Fort Monmouth, NJ 07733-5000.

- SOI identifies problem domain members by finding objects that participate in domain functions;
- SOI models a problem domain function as a set of object transactions; and
- SOI uses a form of partitioning and stimuli to initiate domain functions.

SOI extends the existing scenario-driven approaches to incorporate systematic, repeatable methods. SOI is based on three major concepts:

(1) *object hiding,* which enables the decomposition of a problem domain into levels of abstraction;

(2) *Views,* which capture and describe abstraction levels of a problem domain; and

(3) *View analysis,* which identifies objects that play a role in problem domain functions.

A. OBJECT HIDING

Object hiding extends from aggregation and the aggregate-component relationship. Many times an object will exist solely as a component of another object. A single aggregate may have several parts and each part may itself be an aggregate object. In Object-Oriented Analysis (OOA), the aggregate-component relationship is many times referred to as the whole-part relationship.

We apply Rumbaugh's (6) guidelines to identify objects that are hidden inside other objects. If an object exists solely as the component of another aggregate object, then we say the component object is "hidden" inside the aggregate object.

Object hiding is analogous to information hiding in structured analysis. It enables a problem domain to be expressed at different levels of abstraction. In SOI, each level of abstraction is captured in a View. A View is a model of the problem domain at a particular level of abstraction. Details of a View can be suppressed to lower level Views through object hiding. It is the notion of object hiding and Views that SOI exploits to reduce the complexity of a problem domain and simplify the object identification process.

B. VIEWS

Figure 1 shows a simplified model of a View. A View describes objects and object transactions of a problem domain at a particular level of abstraction. Views contains a set of objects and stimuli that act on those objects. Views also depict transactions that take place between View objects. Symbol **obj** represents a View object. Symbol **t** represents a View transaction.

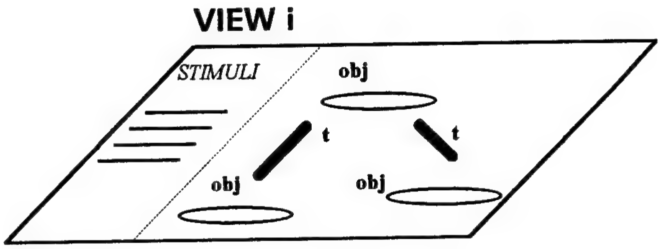


Figure 1.

In SOI, a problem domain is described by a set of Views. Each View models the problem domain at a particular level of abstraction. Views differ in granularity. The top-most View describes a problem domain in its most abstract form. It contains the minimal set of objects and object transactions needed to describe the functionality of a problem domain. Lower level Views contain component objects of the top-most View, and describe object transactions of the top-most View in greater detail. In SOI, each View contains component objects of the previous View, and models object transactions of the previous View in greater detail. Figure 2 illustrates the relationship between domain Views. In figure 2, $View_2$ contains the component objects and transaction details of $View_1$; $View_3$ contains the component objects and transaction details of $View_2$, and so on. To describe the problem domain in its most detailed representation, we require the union of all domain Views:

$$\mathit{MOST\ DETAILED\ REPRESENTATION\ OF\ PROBLEM\ DOMAIN} = \mathit{VIEW}_1\ U\ \mathit{VIEW}_2 \ldots U\ \mathit{VIEW}_{n-1} U\ \mathit{VIEW}_n$$

C. VIEW COMPONENTS

A View is made up of four component types: View objects, object transactions, stimuli, and control mechanisms.

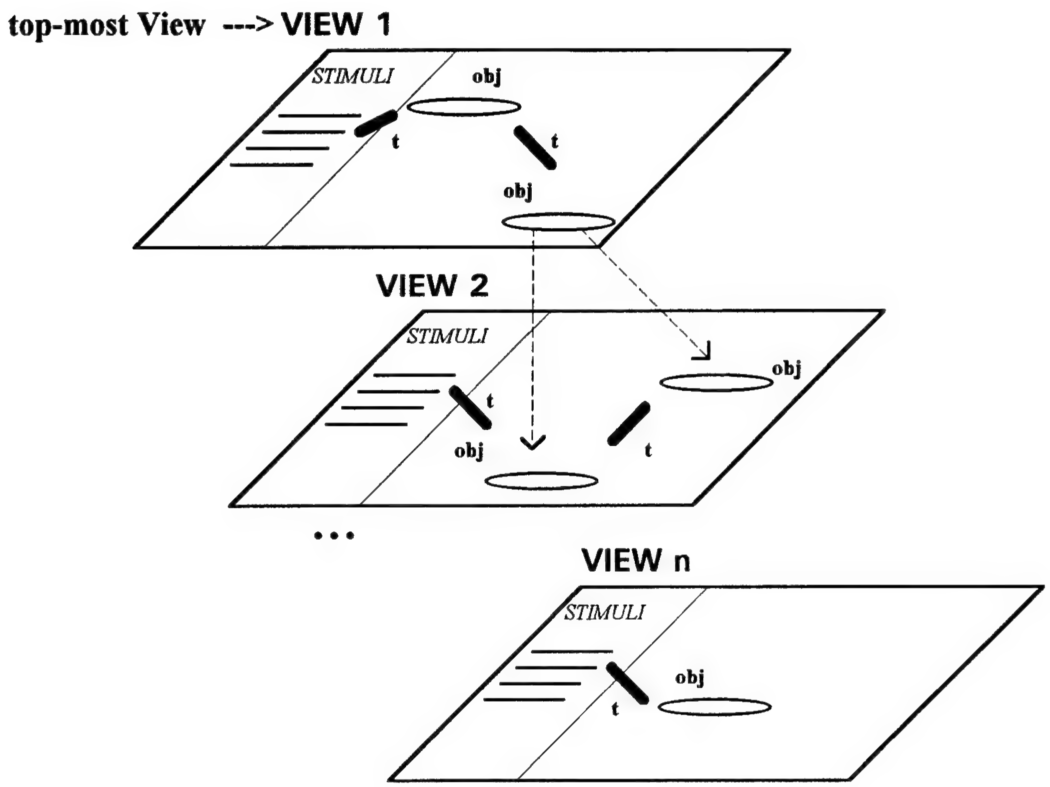


Figure 2.

a. Objects

The following is a simplified model of an object.

```
\OBJECT\
----------------------
|INFORMATION|
---------------------
SERVICES
----------------------
```

Object information is divided into two classifications: attribute information and state information. Attribute information describes specific characteristics of an object. State information describes the overall condition of the object (the object status).

```
\OBJECT\
-------------------
{STATE}
{ATTRIBUTES}
-----------------
SERVICES
-------------------
```

The current value of an object state is defined by stimuli acting on the object and the collection of current values stored in object attributes.

STATE = STIMULI+ COLLECTION_ATTRIBUTE_VALUES

An object can be made of other lessor component objects. When an object, obj1, exists solely as a component of another object, obj2, then we say that obj1 is "hidden" in the services partition of obj2. Consequently, obj2 is visible at a higher level of abstraction than obj1.

```
\obj2\
-----------------------------------------------------------------
{STATE}
{ATTRIBUTES}
-----------------------------------------------------------------
SERVICES:
\OBJECT\     ------------->  \obj1\     <------ HIDDEN
------------  <-----------    ----------          OBJECTS
{STATE}                       {STATE}
{ATTRIBUTES}                  {ATTRIBUTES}
------------                  -------------------
SERVICES                      SERVICES
------------                  ----------------------
```

D. OBJECT TRANSACTIONS

An object transaction is an interaction between a set of initiators and a participant object where the initiators contract that participant's services. We model an object transaction as

$$t = A\text{->}o:s'[\text{->}s''$$

where,

- A is a stimulus which groups a set of initiator actions into a single initiating action;
- o is an object that is acted on by A;
- s' represents an active service task in o before stimulus A; and
- s'' represents a task initiated by o in response to A.

E. STIMULI

Stimuli provide a construct to describe multiple initiating actions in an object transaction. We define two types of View stimuli: external stimuli and internal stimuli. External stimuli are generated by external entities to the problem domain; and act on objects in the top-most View only. Internal stimuli are generated by entities within the problem domain and act on other domain objects.

a. External Stimuli

External stimuli model actions of external entities to a problem domain that cause a state, or attribute change in View objects. We identify external stimuli by modeling the problem domain user interface. To model the problem domain interface, we identify agents within the problem domain that exchange information with the external environment. These agents, termed interface objects, are found by examining all exchange points between the problem domain and the external environment and by extracting the domain entity responsible for information exchange.

We define each type of information that an interface object can extract from the external environment as an input parameter to the problem domain, and represent it as a domain input variable v. The problem domain interface V, is defined as the set of all domain input variables:

$$V = \{v_1, v_2 \ldots v_n\}$$

For each input variable $v_{i,}$ there are m_i states,

$$v_i = \{s_1, s_2, \ldots s_{mi}\}$$

The external stimuli of the problem domain U, is defined as the Cartesian product of all domain input variables.

$$U = \prod_{i=1}^{n} v_i$$

A stimulus, u, is defined as a tuple transition in U. A tuple transition in U denotes a change of state in one or more domain input variables. A stimulus occurs when one or more domain input variables changes state. The following picture illustrates an example tuple in U.

v_1	v_2	....	v_n
m_1	m_2	....	m_n

where,

- n represents the number of input variables to the problem; and
- m_i represents an arbitrary state for v_i.

Each variable in a tuple exists in only one state. A tuple transition, or stimulus can be described as

v_1	v_2	....	v_n
m_1	$m_{2'}\text{->}m_{2''}$	....	m_n

where,

$m_{i'}\text{->}\ m_{i''}$ denotes a state transition in variable v_i.

We can relate an external stimulus of a View to View object transactions by

$$t^i = u^i\text{->}o^i:s'^i[\text{->}s''^i$$

where,

- u^i is a stimulus in View$_i$ which groups a initiating actions of external entities;
- o^i is an object in View$_i$ acted on by u^i;
- s'^i represents an active service task in o^i before stimulus u^i; and

S''^{i} represents a task initiated by o^i in response to u^i.

b. Internal Stimuli

Internal stimuli model actions of objects internal to a problem domain that cause status or attribute changes in other objects. Internal stimuli are generated from state transitions of View objects.

We represent the state of each View object as internal variable g and define G as the set of View internal variables:

$$G= \{g_1, g_2 \cdots g_n\}$$

For each internal variable $g_{i,}$ there are m_i states,

$$g_i= \{g_1, g_2, \cdots g_{mi}\}$$

Internal stimuli of a View is defined as the Cartesian product of all View internal variables.

$$Q= \prod_{i=1}^{n} g_i$$

In a similar manner as external stimuli, we define an internal stimulus q as a tuple transition in Q. We can relate an internal stimulus of a View to View object transactions:

$$t^i = q^i\text{->}o^i{:}s'^i[\text{->}s''^i$$

where,

- q^i is a stimulus in View_i which groups initiating actions of entities inside the problem domain;
- o^i is an object in View_i acted on by q^i;
- s'^i represents an active service task in o^i before stimulus q^i; and
- s''^i represents a task initiated by o^i in response to q^i.

F. CONTROL MECHANISMS

Control mechanisms model two types of objects: objects that coordinate two or more objects during an object transaction; and objects that impose timing constraints on other objects (e.g. time delays). We will discuss control mechanism in greater detail later in the paper.

G. VIEW ANALYSIS

Many researchers (1,2,4,6,7,8) agree that an object that contributes to the functionality of a problem domain qualifies as a domain member. View Analysis identifies objects that contribute to problem domain functions. View analysis finds objects in a manner similar to scenario-based OOA. It models a problem domain function as a set of object transactions, and then identifies objects that participate in the domain functions.

a. View Analysis Notation

Given F, the set of all functions of problem domain D. F can be expressed,

$$F = \{f_1, f_2 \cdots f_k\}$$

where f_j is a domain function and k is the number of functions in D. Function f_j can be expressed in terms of problem domain Views and object transactions:

$$f_j= \{T_{1j}, T_{2j}, \ldots, Tn_j\}$$

where Tij is a set of object transactions in View_i that contribute the functionality of f_j. Also, n represents the number of Views used to model the problem domain. T_{ij} can be represented as

$$T_{ij}= \{t^{ij}_1, t^{ij}_2, \ldots, t^{ij}_{mij}\}$$

where t^{ij} is an object transaction in View_i contributing to the functionality of fj, and m_{ij} is the number of transactions in View_i that contribute the functionality of f_j.

We defined an object transaction earlier in terms of View stimuli:

$$t^i = A^i\text{->}o^i{:}s'^i[\text{->}s''^i$$

We can relate transaction t^i to function f_i by

$$t^{ij} = A^{ij}\text{->}O^{ij}{:}S^{ij'}\ [\text{->}S^{ij''}$$

where,

A^{ij} is a stimulus that acts on a set of objects in View_i, and initiates transaction that contributes to the functionality of f_j;

O^{ij} is a subset of View$_i$ objects that is acted on by A^{ij} and initiates service task that contribute to the functionality of f_j ;

$S^{ij}{}'$ is a subset of object services available in View$_i$, that are active before stimulus A^{ij}; and

$S^{ij}{}''$ is a subset of object services available in View$_i$, that are initiated by O^{ij} in response to A^{ij}, and contribute to the functionality of f_j.

We can model function *fj* in terms of object transactions:

$$\begin{aligned} f_j = \; & A^{1j}{}_1\text{->}O^{1j}{}_1:S^{1j}{}_1{}'[\text{->}S^{1j}{}_1{}'', \\ & A^{1j}{}_2\text{->}O^{1j}{}_2:S^{1j}{}_2{}'[> S^{1j}{}_2{}'' \\ & ,\ldots, A^{1j}{}_{m1j}\text{->}O^{1j}{}_{m1j}:S^{1j}{}_{m1j}{}'[\text{->} S^{1j}{}_{m1j}{}'', \\ & A^{2j}{}_1\text{->}O^{2j}{}_1:S^{2j}{}_1{}'[\text{->} S^{2j}{}_1{}'', \\ & A^{2j}{}_2\text{->}O^{2j}{}_2:S^{2j}{}_2{}'[\text{->} S^{2j}{}_2{}'' \\ & ,\ldots, A^{2j}{}_{m2j}\text{->}O^{2j}{}_{m2j}:S^{2j}{}_{m2j}{}'[\text{->} S^{2j}{}_{m2j}{}'', \\ & \qquad \ldots. \\ & A^{nj}{}_1\text{->}O^{nj}{}_1:S^{nj}{}_1{}'[\text{->}S^{nj}{}_1{}'', \\ & A^{nj}{}_2\text{->}O^{nj}{}_2:S^{nj}{}_2{}'[\text{->} S^{nj}{}_2{}'' \\ & ,\ldots, A^{nj}{}_{mnj}\text{->}O^{nj}{}_{mnj}:S^{nj}{}_{mnj}{}'[\text{->} S^{nj}{}_{mnj}{}'', \end{aligned}$$

where,

m_{ij} is the number of object transactions in View$_i$ that contribute to the functionality of f_j, and n is the number of Views used to model the problem domain.

We identify an object *o* as a member of problem domain *D*, if *o* is defined in f_j, and f_j is function in *D*. Object *o* is defined in f_j if *o* executes a state change to contribute to the functionality of f_j. We can identify all problem domain members with the expression:

$$o|\forall o(\exists f \subseteq F \Lambda o \in f).$$

b. Application Of View Analysis

We apply View analysis to identify objects that participate in domain functions. View analysis begins with the top-most View and proceeds in a top-down manner until processing the most detailed View of the problem domain. Analysis begins with the top-most View because all external stimuli in the problem domain are defined in the top-most View, and SOI assumes that all domain functions are initiated through external stimuli.

Analysis proceeds by activating each View stimuli, and studying the resulting object transactions. Each transaction initiates service tasks in one or more View objects. Task initiation is reflected in the status variable of each object by a status transition. We can identify domain members by applying the concepts of the previous section:

If an object reflects a status transition as a result of a View stimuli, then that object qualifies as a member of the problem domain.

A status transition in an object generates an internal stimulus. An internal stimulus can affect other View objects, and can also affect components of objects from which the stimulus originated. Internal stimuli demonstrate the transitivity of a domain functions. The initiation of a domain functions generates a set of object transactions that generate internal stimuli, which in turn generate a set of object transactions, and so on. A function concludes when the resulting internal stimuli of a set of objects transaction do not produce another set of object transactions. Figure 3 illustrates the phenomena of transitivity:

External stimulus *u1* initiates object transactions *t1* and *t2* in *View₁*. As a result of *t1* and *t2,* objects *obj1* and *obj3* generate internal stimulus *q1*. Stimulus *q1* initiates transaction *t3* in *View₁.* In addition, *q1* initiates transactions *t1* and *t2* in *View₂* because object *obj1* and *obj2* of *View₂* are component objects of *obj3* in the previous View. Note that stimuli *q2* and *q3* failed to generate object transactions. Internal stimuli do not always result in transactions because they may not impact other View object. Also, an internal stimulus can not impact the next View if it originates from objects that do not have component parts.

A problem domain function can be modeled as a sequence of View stimuli. In SOI, we assume that every domain function is initiated by some external stimulus u. External stimulus u initiates a set of object transactions

$$u\text{->}\{o_1:s_1{}'\text{->}s_1{}'', o_2:s_2{}'\text{->}s_2{}'', \ldots, o_m:s_m{}'\text{->}s_m{}''\},$$

where,

m is the number of object transactions generated by u.

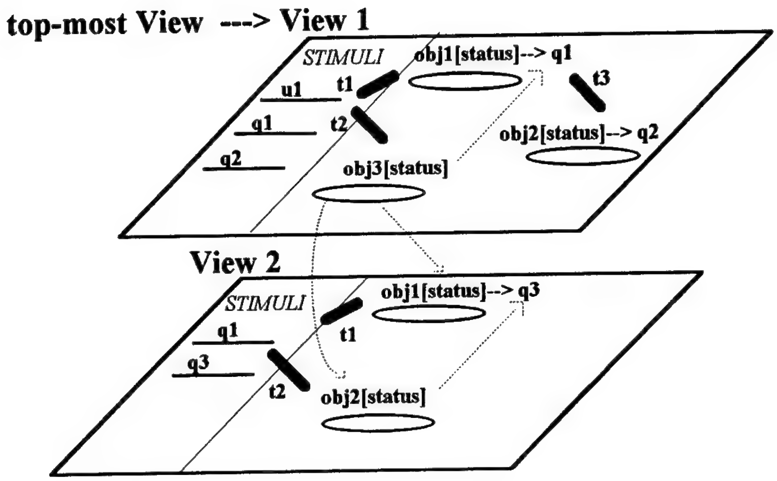


Figure 3.

Transactions generated by u, initiate a set of object state transitions which, in turn, generate an internal stimulus Q,

$$q = \{s_1\text{'->}s_1\text{''}, s_1\text{'->}s_1\text{''}, \ldots, s_m\text{'->}s_m\text{''}\}.$$

Internal stimulus q can also generate a set of object transactions, which, in turn, generate an internal stimulus, and so on.

We can see in figure 3 that function transitivity is propagated through problem domain View through internal stimuli and components of the aggregate objects that generate the stimuli.

View analysis initiates a domain function in the top-most View and then captures the resulting object transactions as the function propagates through Views of the problem domain. During View analysis we examine each View separately. We apply View stimuli and then monitor each View object for a status transition. If an object reflects a status change as a result of any View stimuli, then the object qualifies as a member of the problem domain. If an object never changes status, all View stimuli have been applied to the View, then the object is eliminated from the View and the problem domain because it does not contribute to domain functions. We continue View analysis until all candidate objects have qualified as domain members, or been eliminated.

Rules Governing the Application of Views

Several rules apply to the application of Views:

1. A View must contain at least two objects.
2. Only objects visible at a common level of abstraction can be contained in a common View.
3. Only objects acted on by a View's external event, or involved in an object transaction can be considered a View member.
4. A View's hidden objects must be "pushed down" to a lower level View.
5. An object must fulfill the following criteria to be qualify as View and problem domain member:

   o The object can not be hidden in another member's services partition; and
   o At least one View stimuli must act on the object.

Conclusion

We have applied our method to a large number of problem domains, and found that it is repeatable and systematic. Programmatically, we apply it in the following nine steps:

o Generate Problem Statement
o Extract and Classify Nouns
o Associate Candidate Objects with Remembered Events and Things
o Model System Interface and Identify Use Cases
o Constrain Candidate Object States
o Construct View
o Simplify View through Object Hiding
o Simplify View by Event Analysis
o Process Next View

Presently, we are developing a tool to automate our method.

References

[1] G. Booch, Object-Oriented Design with Applications, *The Benjamin Cummings Publishing Company, Inc.*, 1991.

[2] P. Coad and E. Yourdon, Object-Oriented Analysis, *2nd ed., Yourdon Press*, 1991.

[3] A. Goldberg and D. Rubin, Smalltalk-80: The Language and Implementation, *Addison-Wesley*, 1983.

[4] I. Jacoson and F. Lindstrom, Re-engineering of Old Systems to an Object-Oriented Structure, *In ACM OOPSLA'91 Conference Proceedings*, October 1991.

[5] D. L. Parnas and P. C. Clements, A Rational Design Process: How and Why to Fake It, *IEEE Transactions in Software Engineering*, February 1986.

[6] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Object-Oriented Modeling and Design, *Prentice-Hall International*, 1991.

[7] S. Shlaer and S. J. Mellor, Object Lifecycles: Modeling the World in States, *Yourdon Press*, 1992.

[8] R. Wirfs-Brock, B. Wilkerson, and L. Wiener, Designing Object-Oriented Software, *Prentice-Hall International*, 1990.

About the Authors:

Dr. Monroe is the Chair and Professor of Computer Science at NC A&T State University. His teaching and research interests include software engineering, object-oriented design and software reuse. Dr. Monroe may be reached by e-mail as monroe@garfield.ncat.edu.

Dr. Yu is an Assistant Professor of Computer Science at NC A&T State University. Her teaching and research interests are in the fields of software engineering, software reuse and concurrent programming languages. Dr. Yu may be reached by e-mail as cshmyu@garfield.ncat.edu.

ADA VERSUS PASCAL FOR SOFTWARE ENGINEERING EDUCATION AND PRACTICE

Abdullah Al-Dhelaan and Jagdish C. Agrawal
Computer Science Department, King Saud University

Abstract

Both Ada and Pascal offer strong support for teaching Data Structures allowing specification and implementation of abstract data types in a natural manner. However, we found that the students using Pascal, have a tendency to look at and use the implementation details of a data structure, and thereby violate the scope provided in the specification of that data structure.. However, Ada, through separation of package specification and package body, and also through private and limited private type, forces the students to limit themselves to the scope provided in the specification of a data structure. In this paper, we provide interesting examples from our experience to make this point. There are several nonstandard versions of Pascal in the market, which to different degrees do provide limited support in this area, but the nonstandard nature of those languages raises much higher level questions of portability, inter-operability and maintainability. Ada has eliminated proliferation of versions by simply requiring strict adherence to standards. Therefore, we will limit our discussion to only ANSI Ada and ANSI Pascal.

Students as well as practitioners of Software Engineering need tools that support Software Engineering Principles and Goals. Above all, their tools must comply with ANSI or ISO standard. Further, such tools should offer support for implementing these principles so that the goals of software engineering can be achieved. Compared to Ada, Pascal exhibits weakness in implementing information hiding, localization, and confirmability.

1.0 Introduction

A programming language, as a tool for education and practice of software engineering should provide means of applying software engineering principles as outlined in a classical paper by Ross, Goodenough and Irvine.[1] Well known advocates of structured programming: Dahl, Dijkstra, and Hoare,[2] and Wirth[3] recommend problem refinement using a restricted set of sequencing structures to document the design. It is important that in such problem refinement, when one abstracts a real world object as a data structure, say **apple**, then at a later stage, there should be no temptation to use it as an **orange**. In a later section, we will show that Pascal students can and indeed, do such drastic violations of abstraction principle.

Ledgard and Marcoty[4] developed a model to describe a typical programming task that goes a step forward from the structured programming. Ledgard's approach looks at the real world objects with associated set of appropriate operations in the problem space. Also, in the problem space, there are real-world algorithms that transform the problem objects into resulting solution objects. In Ledgard's approach, the solution space parallels the problem space. In the solution space, programming language provides a tool with which the programmer can abstract the objects of the problem space and implement the abstraction in software solution. It is therefore, important that the programming language easily allow its user to design and implement these abstractions. The implementation should be such that it does not allow violation of the specification for which the abstraction was implemented. For example, if one builds an abstract data type, say MyStack of StudentRecords is implemented in Pascal, either with arrays of StudentRecords or by using a linked list of nodes that contain StudentRecords. The abstraction provides capabilities to InitializeStack, Pop, Push, and CopyTop. If the user of these stacks has the need to search for a particular record, he should not be allowed to violate the specification of the abstraction stack. However, both Pascal implementations do allow violations as we will discuss in the next section.

Dey and Mand[5] conducted a survey of four year programs in Computer Science, and from their survey, concluded: "... Pascal continues to be the primary instructional language, used in almost three-fourths of all universities and four year college. ... " As pointed out above, while teaching both Pascal and Ada, we collected data on how well the specifications were enforced, and how well the principles of software engineering were followed. Based on such data, we have derived some opinions about the merits of Ada versus Pascal for teaching the principles that we want to teach in a Computer Science curriculum.

2.0 Package facility of Ada

Packages are among Ada's fundamental program units. This is not the case with ANSI Pascal, and using nonstandard tools is against our philosophy of teaching software engineering, hence we collected no data on nonstandard languages that use "Pascal" as part of their names.

Pascal procedures and functions could be considered somewhat closer in structure, structure to packages in Ada, because they allow encapsulation of logically related computational resources. With Pascal's rules on the scope, these structures do offer abstraction, modularity, and localization with limited amount of information hiding. However, a data type declared locally, inside a procedure is available only inside it, and hence is of little use to clients outside. Pascal's procedures offer functional services by performing the tasks of those procedures, but do not share their local objects with the outside world.

Ada's packages allow encapsulation of logically related objects and methods. A package in Ada consists of two parts: (1) package specification, and (2) package body. Package specification is a tool for describing an Abstract Data Type (ADT) with a specification of its form and functionality to the outside world of users or clients. The package body provides the implementation of these ADT's, but these implementation details are hidden from the users and clients. Ada provides separate compilation of package specification and package body. Thus, Ada provides strong support for two important principles of software engineering:

- Abstraction, and
- Information Hiding

While Ada's package specification provides its ADT's to the clients, while hiding the implementation detail of the ADT, Pascal's information hiding about the ADT inside a procedure is absolute, and does not make any part of that abstraction accessible to clients outside.

Ada's package specification becomes an interface with the outside world. If any changes are made in the implementation (code in the package body) of an ADT, the interface with the outside world remains intact. ANSI Pascal allows use of ADT's of a procedure only to clients subordinate to that procedure and makes the implementation detail of the ADT accessible to them. Thus, in Pascal, it is possible that any changes in the implementation of an ADT can create a long chain of reaction requiring corresponding changes in all client users of that ADT.

3.0 Ada's Exception Handling

Ada allows declaration as **exception** of an event that can cause suspension of normal execution of the program. Further, it allows declaration of user defined exceptions, in addition to several predefined exceptions such as:

- CONSTRAINT_ERROR
- NUMERIC_ERROR
- PROGRAM_ERROR
- STORAGE_ERROR
- TASKING_ERROR

While an exception is a condition, not an object, it can be declared anywhere the declarations of objects are permissible. The detection of exceptional conditions does impose a small runtime overhead, but it assists in enforcing quality assurance, by promoting increased understandability, reliability, and maintainability. Through its exception handling power, Ada provides the programmer with the means to intercept exception condition and pass the control to the exception handler. The alternative provided in most other languages, including ANSI Pascal is suspension of the processing of the program and returning the control to the operating system. For a run time system, such as auto pilot, or navigation system, this would be totally unacceptable.

Students and practitioners of Software Engineering need to learn to plan for unexpected events (exceptions) because we cannot predict when such events will occur. Total system shut down by such unexpected events is unacceptable, particularly in applications such as a software system guiding and controlling a nuclear power plant, or a satellite, or a manned aircraft. Reliability is an important goal of software engineering, and the ability to control unexpected events is an important factor in improving reliability.

4.0 Ada's Generic Program Units

Ada also provides templates for algorithms with multiple uses. This is done with the use of generic packages and subprograms. Users of such generic templates simply create a specific instance of the generic unit by making a local copy through a process called instantiation. For instantiation, the user provides an identifier that names the regular program unit and maps generic parameters with actual parameters similar to the matching between actual and formal parameters in Pascal.

Students and programmers alike can and should learn about increasing reusability by building generic objects. For example, a generic Stack object will be far more reusable, than MyStack of StudentRecord mentioned in section 1.0 above:

```
generic
   type type_of_object is private;
   max_size : natural := 5000;
package generic_stack is
   stack_full : exception;
   stack_empty: exception;
   procedure push (object : type_of_object);
   function size_of_stack return natural;
   function copy_top return object_type;
   procedure pop;
end generic_stack;
```

User could then instantiate generic_stack to create my_stack with a statement like:

```
package my_stack is new generic_stack (
        type_of_object => studentRecord_type,
        max_size => 100);
```

In the example above, the essence of stack was created with a very high level of abstraction, and hiding the type of objects being stacked and the maximum size. Lacking the capability of generic object handling, ANSI Pascal appears weak here.

5.0 Importance of Private Types

Because the data type declarations made in a package specification are visible, the user might be tempted to use the specific knowledge about the structure of that object. For example, if a stack_of_record type is declared as a record containing an array of records, and the integer representing the number of records in the stack, the beginner student is tempted to take short cuts when searching for a particular record in the stack. This defeats the intent of the teacher who wants to teach an algorithm requiring popping from the stack and pushing on to a temporary stack until the particular record is found and then transferring back from the temporary stack back to the original stack. Further, the student taking short cuts tend to make their systems less modifiable.

Ada provides a means of hiding the structure of objects declared in a package specification through the use of private and limited private type. Pascal appears weak in this area. The second author of this paper collected some empirical data from two mid-term examinations he gave to his eighty four students in the Structured Programming class (using Pascal). The data came from three questions. One required the students to declare a linked list of Student_Records and create a procedure to insert a given Student_Record in the linked list in increasing order of Student_Record.Key. The data was to come from an array of these records in which the records were in random order. About twenty five percent of the students who didn't want to put up with the challenge of linked list traversal to find the correct location and then the steps needed for the insertion, went around the question. They did a quick sort or a bubble sort of the array using a temporary array and then simply attaching one record to the tail at a time. Another question that required search for a record in a dynamic stack (implemented using a linked list type structure). Again about thirty percent went around the question and did a traversal of the linked list. Another question, in which the students were provided the procedure to insert student records in a linked list, in increasing order of Student_Record.Key, and were asked to sort an array of Student_Records by making calls to the given procedure. About thirty five percent of the students chose not to use the given insert in order procedure at all. They went ahead and supplied either a bubble sort, or a quick sort, or a selection sort. What this data shows is that it is very important that the student of software engineering be given the tools that give them little temptations to take short cuts and violate the principles of software engineering.

6.0 Conclusions

Ada and ANSI Pascal are both excellent for teaching many programming principles. However, if Software Engineering principles and goals are being taught for practice and use, ANSI Pascal appears rather weak. The success, popularity, and proliferation of nonstandard versions of Pascal that claim to offer some of the Ada like features further confirms this. Now that Ada has been around for about a decade, and there are good text books and compilers, universities will be doing their software engineering students a big favor by adopting Ada in their programming course sequence.

7.0 References

1. D. T. Ross, J. B. Goodenough, and C. A. Irvine, "Software Engineering: Process, Principles, and Goals," IEEE Computer, May 1975.

2. O. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Notes on Structured Programming, Academic Press, 1972.

3. N. Wirth, "Program Development by Step-Wise Refinement," Communications of the ACM, Volume 14, Number 4, pp. 221-226.

4. H. Ledgard and M. Marcotty, The Programming Language Landscape, Science Research Associates, Chicago, IL, 1981.

5. S. Dey and L. R. Mand, "Current trends in Computer Science Curriculum: A Survey of Four-Year Programs," SIGCSE Bulletin, Volume 24, Number 1, March 1992.

8.0 Authors' Biographical Sketches

Name: Abdullah Al-Dhelaan
Address: P. O. Box 51178, Riyadh-11543
Saudi Arabia
Email: F60C018@SAKSU00.BITNET

Dr. Abdullah Al-Dhelaan is currently the Vice Dean of Admission and Registration and Assistant Professor of Computer Science at King Saud University, Riyadh. Prior to assuming the duties of Vice Dean, he was the Chairman of the Computer Science Department at King Saud University. Dr. Al-Dhelaan has also served as the Director of National Information Center, Ministry of the Interior., Kingdom of Saudi Arabia. Dr. Abdullah Al-Dhelaan is on the faculty of the College of Computer & Information Science, King Saud University, since early 1989. His areas of research are Software Engineering, Parallel Processing, Distributed Systems, and Interconnection Networks.

Name: Jagdish C. Agrawal
Address: P. O. Box 51178, Riyadh-11543
Saudi Arabia
Email F60C029@SAKSU00.BITNET

Dr. Jagdish C. Agrawal is Professor of Computer Science at King Saud University, Riyadh, Saudi Arabia, since January 1990. He has served as Department Chair at Purdue University, Ft. Wayne (1981-82), Division Chief at the Army Institute for Research in Management, Information & Computer Sciences (AIRMICS), Department of Defense (1983-86), and as Computer Science Department Chair at Embry Riddle Aeronautical University, Daytona Beach, FL (1986-90). He also served as Professor of Mathematics and Computer Science at California University of PA (1969-86) from where he took early retirement to become Professor Emeritus. The software engineering projects that Dr. Agrawal worked on at AIRMICS included ARS, STEP, SRT, Follow on Evaluation of PATRIOT etc. He has over seventy publications.

ADA VERSUS MODULA-2: A PLAN FOR AN EMPIRICAL COMPARISON OF THEIR EFFECTIVENESS IN INTRODUCTORY COMPUTER PROGRAMMING COURSES

H. K. Dai and K. E. Flannery
Department of Computer Science
University of North Dakota
Grand Forks, North Dakota 58202
U. S. A.

Abstract

The language Ada has been used widely and successfully in introductory computer programming courses taken predominantly by computer science majors, but has enjoyed limited success as a language for teaching programming to non-computer-science majors. This paper discusses the motivation and plan for using Ada in introductory programming courses for students in a college of aerospace sciences, and details a plan for an empirical study of the relative effectiveness of using Ada versus Modula-2 in introductory computer programming courses. The study will try to answer the question, "Is Ada a more effective pedagogical tool for teaching computer programming than Modula-2?" The proposed study is being implemented at a medium-sized university in the northern plains region.

1 Introduction

The language Ada has been used widely and successfully in introductory programming courses designed for computer sciences majors (see [Eva85], [FS90], [RZP$^+$88], [WL89], and [TE87]), however, there is little indication whether or not using Ada would be as effective in introductory computer programming courses taken predominantly by non-computer-science majors. We discuss choosing Ada for such courses at the University of North Dakota.

1.1 The Department of Computer Science and the Center for Aerospace Sciences

The University of North Dakota is a state-assisted institution of higher education located in Grand Forks, a town on the border of North Dakota and Minnesota. The Department of Computer Science at the University is part of the Center for Aerospace Sciences and offers undergraduate degrees and a graduate degree in computer science. The undergraduate program is accredited by the national Computer Science Accreditation Board. The department currently has 10 faculty members and approximately 200 undergraduate and 25 graduate students majoring in computer science.

As part of the Center for Aerospace Sciences, which includes the Departments of Atmospheric Sciences, Space Studies, and Aviation (the largest, with around 400 students), the Department of Computer Science has a unique role, providing instruction for a wide range of scientific and practical computing applications.

1.2 The Introductory Programming Courses

CSci 160 and CSci 161, "Computer Programming I and II," are each mandatory, four-hour one-semester introductory courses with a one-hour weekly lab, and are usually the first two programming courses that computer science majors take. CSci 160 also serves to fill a required programming component for other departments in the Center for Aerospace Sciences, and hence has a large enrollment each semester: over 200 students, with only about 20% computer science majors.

The introductory programming course teaches structured programming methodology using a variety of problems and applications. CSci 160 also has a closed laboratory component where the class meets once each week to perform a one- to two-hour programming exercise. CSci 161 can be viewed as a continuation of its prerequisite CSci 160, where the accent is on medium-scale software design, and the tools for reasoning about the correctness, modularity and efficiency of programs. It, too, contains a closed laboratory component. The size of the CSci 161 class is usually between 30 and 40 students per semester, 70% of which are computer science majors.

1.3 Shifting to Ada

The authors have acquired a course-development grant from the the Defense Advanced Research Projects Agency and the Ada Joint Program Office (1993) for designing and implementing a conversion of the introductory sequence in computer programming and data structures to use the Ada language. This sequence will be an integrated two-semester introduction to programming in Ada with emphasis on correctness, style, and efficiency. The courses will fully exploit elementary features of Ada, such as variables, expressions, and statements, and will introduce more advanced features, such as subtypes, subprograms, overloading, exception handlers, packages, and generics.

The courses are to be offered concurrently with ones using Modula-2 as the programming language, and will utilize a Sun 670 (4 processors) running a validated Ada compiler. The set of collective observations on the students enrolling in these concurrent classes provide the raw material of a statistical investigation on the effectiveness of using Ada versus Modula-2 in introductory computer programming courses.

2 Motivation for Adopting the Language Ada

The language evaluations and comparisons for Modula-2 and Ada in the contexts of system programming, software engineering, and pedagogy have been studied extensively in the literature (see for examples, [ATW83], [Eva85], [EP85], [Fel90], [TE87], and [WL89]). The following have substantiated our decision to gradually move away from Modula-2 as our principal language into Ada.

2.1 Superior Pedagogy

The clearest reason for adopting Ada is that it is, based on our academic experience and that of others, intrinsically a superior pedagogical language. The commitment of Ada to structured programming and strong typing allow it to be used in Pascal-like introductory courses. Ada also represents the state of the art in large-scale, object-based software construction, and hence provides a contemporary framework for a wide range of intermediate and advanced topics, including software engineering methodology, system programming, database design, and real-time processing.

Ada well supports a rigorous introduction to computer science via programming. Algorithmic is supported by the Pascal-like features of Ada and additional features such as aggregate initialization and subtypes. Abstraction and program design and analysis at the introductory level are supported by features such as subprograms, overloading, exception-handlers, and packages. T. Frank and J. Smith [FS90] provide further rationale of the adoption of Ada as a CS1-CS2 language.

For a data structures course emphasizing the specification and implementation of abstract data types and algorithms, Ada also provides well. The package construct of Ada allows the separation of specification from implementation, and allows implementation details to be hidden from the user. Generics in Ada (parametrized program units) allow many abstract data types to be defined in a general way, with respect to a base type; for example, one can define list data types for arbitrary element type. The Ada exception mechanism provides controlled and efficient processing of error conditions, which is useful in specifying and implementing abstract data types, as well as in using them. M. Feldman [Fel90] details some language-to-language comparisons which support Ada as a data-structures language.

Ada would be a useful language in many advanced courses. The task and task synchronization primitives in Ada could serve well in operating systems courses and courses involving other real-time applications. The program library concept makes Ada inherently a tool-builder's language, and would fit well in database and software engineering courses.

2.2 Proliferation of Ada in Industry

With the fused efforts of academia, industry, and government, Ada is likely to proliferate throughout the computing community and become a bona fide production language with widespread use in industry. Its support for concurrency and the high degree of portability of its programs makes Ada a powerful tool for students, now and upon graduation, whether their needs focus on immediate employment or further formal education.

2.3 Stability and Extensibility of the Programming Environment

Among the goals of the Ada Programming Support Environment (APSE) requirements, besides providing tools tailored to the language Ada, are (1) the ability of an APSE to grow or to be customized for a particular software project, and (2) the ability for programmers familiar with one APSE to use another similar APSE with minimal retraiiing. Hence, the ability to create, refine, reuse, and extend software components is guaranteed, from year to year, project to project, in any Ada environment in which a programmer works. The requirements also guarantee the openness necessary for fostering new software methodologies, which in turn will lead to new APSE tools.

2.4 Increasing Demand from Our Community

We have been aware, since our appointments in our Department, that the students are increasingly choosing Ada in their projects in file structures and compiler construction. In fact, the textbook for our recently-offered file structures course was on using Ada-like pseudocode ([MP90]).

Another concern is that our University is the only institution in the state of North Dakota and the northwestern part of Minnesota which currently offers a course in computer programming in Ada. Given our geographical location and student demographics, a formal education introducing the language Ada is unavoidably needed.

2.5 Benefit to Our Co-operative System

Our introductory sequence to computer science and data structures normally enroll a balanced mixture of aviation science, computer science, engineering, and mathematics majors. The University of North Dakota has a moderate size of co-operative students in the Departments of Aviation Science, Computer Science, and Engineering. Proficiency in a language such as Ada should benefit many of our co-operative education students in seeking employment with the government or aerospace industry.

3 Expectations

After offering the trial run of the Ada introductory course sequence, we expect both computer science majors and other aerospace science students completing the sequence to exhibit higher programming proficiency than their Modula-2 counterparts. Specifically, we predict that students completing the Ada sequence will be able to write larger programs (since Ada has more powerful constructs than Modula-2), write programs containing fewer bugs (since Ada's type system is richer and more consistent than in Modula-2), write more efficient programs (Ada's exceptions and generics promote time and space efficiency), and will be able to take better advantage of previously written code (since Ada forces programs to be entered in libraries).

We construct an experiment whose outcomes have some bearing on the hypothesis of interest: the language Ada is a more effective pedagogical tool for teaching computer programming than Modula-2. A widely used formalization of statistical hypothesis testing, due to Neyman and Pearson and some related notions [Leh59], will be employed to test the statistical consistency of the outcomes of the experiment with the hypothesis.

4 The Experiment

The data used in the study will be obtained from selected questions on midterm and final examinations for the introductory course sequences, CSci 160 and 161 (Ada), and CSci 160 and 161 (Modula-2) (Computer Programming I and II, respectively). The questions selected will be non-language-specific programming problems, such as, "Write a program to read batting statistics and to print a batting average," or, "Write a procedure to find the mean, median and mode of an array of test scores." The questions will of course be the same in both Ada and Modula-2 examinations.

Our choice to use test questions as a basis for sampling instead of programming assignments performed out of class seems necessary. The reality is, students collaborate, and it would not be clear whether our samples were indeed distinct. Ideally, we would sample supervised, out-of-class assignments, and although the laboratory period is one such opportunity, there is not enough similarity in the lab exercises in the two course sequences to be useful for comparison.

The sample test question answers will be taken from students in both Ada and Modula-2 classes which

1. have a declared major in a science, mathematics, or engineering department,
2. have not taken Programming I or II before, and
3. have an attendance record of at least 75%.

At the risk of seeming prejudicial, we feel the restrictions are necessary to offset the difference in aptitudes of the Ada and Modula-2 students. Our Programming I class historically contains a substantial subpopulation of unwilling participants from disciplines typically that are non-mathematical in nature, and we feel there will be a larger percentage of them in the Modula-2 section. Conversely, due to a University policy allowing students to retake a course, even if they have passed it, we expect some Modula-2 students to have the unfair benefit of seeing the material before. And, it is always the case that some students stop attending the class, or attend only sporadically. Whether this is due to boredom or just "giving up" is immaterial since we seek to sample only class participants.

A total of 10 questions will be given (5 in each), and when graded, will provide the data for the statistical analysis. In the Computer Programming I course, one question will be taken from the first midterm examination, two from the second midterm, and two from the final examination. The first question will be a coding fragment, the next three will be complete subprograms, and the fifth will be a complete program. In the Computer Programming II course, two questions will come from each of the midterm examinations, and will involve writing a module/package. The last question (question 10) will come from the final examination and will be a complete, non-trivial program requiring the application of structured data types and modules/packages. The relative weights of the questions will depend on how elaborate the solution must be, and hence how much time must be devoted to it. A rough estimate of the question weights appears below:

Question	Course (I/II)	Examination	Points
1	I	1*st* Midterm	10
2	I	2*nd* Midterm	20
3	I	2*nd* Midterm	20
4	I	Final	20
5	I	Final	30
6	II	1*st* Midterm	20
7	II	1*st* Midterm	20
8	II	2*nd* Midterm	30
9	II	2*nd* Midterm	30
10	II	Final	40

To ensure uniformity, each question will be graded twice, once by each investigator, and the final point is determined by the average. Partial credit for programs or pieces of code will be assigned using the following rough guidelines:

Demonstrated understanding of the problem:	20%
Correctness of approach (algorithm):	50%
Correctness of implementation:	20%
Efficiency of solution:	10%

5 The Statistical Analysis

Let $X_1, X_2, \ldots, X_m$ and $Y_1, Y_2, \ldots, Y_n$ be the point measurements of the students enrolling in CSci 160/161 (Ada) and CSci 160/161 (Modula-2), respectively. For quantitative measurements such as performance points, we may assume that $X_1, X_2, \ldots, X_m$ and $Y_1, Y_2, \ldots, Y_n$ are independent samples from the normal populations, $\mathcal{N}(\mu_1, \sigma^2)$ and $\mathcal{N}(\mu_2, \sigma^2)$, respectively.

The assumption of normality is based on a mixture of mathematical convenience and the impression that, in practice, most quantitative measurements or some transform are approximately normal. The assumption that the variances of the two populations are equal corresponds to the requirement that the two pedagogical tools have "constant effect." That is, a student would have had y points were the student in the CSci 160/161 (Modula-2) classes, the student's points when enrolling in CSci 160/161 (Ada) classes is $x = y + (\mu_1 - \mu_2)$. Note that neither of these assumptions is perfectly justifiable.

Our study will be testing the hypotheses

$$H : \mu_1 - \mu_2 \geq 0 \text{ and } H' : \mu_1 - \mu_2 = 0$$

versus their alternatives

$$K : \mu_1 - \mu_2 < 0 \text{ and } K' : \mu_1 - \mu_2 \neq 0,$$

respectively. Uniformly most powerful (UMP) unbiased tests exist for the hypotheses H and H' (see [Leh59]).

The rejection regions of the UMP unbiased α-level tests for testing the hypotheses H and H' are given by

$$\begin{aligned} t(X,Y) &\geq C_H, \text{ and} \\ |t(X,Y)| &\geq C_{H'}, \end{aligned}$$

respectively, where the statistics

$$t(X,Y) = \frac{(\bar{Y} - \bar{X})/\sqrt{\frac{1}{m} + \frac{1}{n}}}{\sqrt{(\sum(X_i - \bar{X})^2 + \sum(Y_j - \bar{Y})^2)/(m+n-2)}},$$

whose distribution is Student's t-distribution, and the constants C_H and $C_{H'}$ are determined by

$$\begin{aligned} \int_{C_H}^{+\infty} t_{m+n-2}(z)\,dz &= \alpha, \text{ and} \\ \int_{C_{H'}}^{+\infty} t_{m+n-2}(z)\,dz &= \frac{\alpha}{2}, \end{aligned}$$

respectively.

References

[ATW83] M. Augenstein, A. Tenenbaum, and G. Weiss. Selecting a primary programming language for a computer science curriculum: PL/I, Pascal and Ada. *ACM SIGCSE Bulletin*, 15(1):148–153, 1983.

[EP85] H. Evans and W. Patterson. Implementing Ada as the primary programming language. *ACM SIGCSE Bulletin*, 17(1):255–265, 1985.

[Eva85] H. Evans, et al. Ada as a primary language in a large university environment. In *Proceedings of the Third Annual National Conference on Ada Technology*, March 1985.

[Fel90] M. Feldman. Teaching data structures with Ada: an eight-year perspective. *ACM SIGCSE Bulletin*, 22(2):21–28, 1990.

[FS90] T. Frank and J. Smith. Ada as a CS1-CS2 language. *ACM SIGCSE Bulletin*, 22(2):47–51, 1990.

[Leh59] E. Lehmann. *Testing Statistical Hypotheses.* John Wiley & Sons, Inc., 1959.

[MP90] N. Miller and C. Petersen. *File Structures with Ada.* The Benjamin / Cummings Publishing Company, Inc., 1990.

[RZP+88] A. Radensky, E. Zivkova, V. Petrova, R. Lesseva, and C. Zascheva. Experience with Ada as a first programming language. *ACM SIGCSE Bulletin*, 20(4):58–61, 1988.

[TE87] W. Tam and M. Erlinger. On the teaching of Ada in an undergraduate computer science curriculum. *ACM SIGCSE Bulletin*, 19(1):58–61, 1987.

[WL89] L. Winslow and J. Lang. Ada in CS1. *ACM SIGCSE Bulletin*, 21(1):209–212, 1989.

The Authors

H. K. Dai is an Assistant Professor at the University of North Dakota, Grand Forks, North Dakota. He received his Ph.D. in Computer Science from the University of Washington (Seattle, Washington) in 1991, under the direction of Richard Ladner. His research interests include computational complexity and parallel computation.

K. E. Flannery is an Assistant Professor at the University of North Dakota, Grand Forks, North Dakota. He received his Ph.D. in Computer Science from the Virginia Polytechnic Institute in 1989 under the direction of J. J. Martin. His primary interests are in programming language semantics, translation, and related algorithms, and has published articles on domain theory, type checking, and automatic semantic interpretation.

A Note on Teaching Abstract Data Type Using Ada and C++

Huei–Chung Chu, Ke–Hsiung Chung, Bau–Hong Yuan
Department of Information Management
National Defense Management College
Taipei, Taiwan, R.O.C.

Abstract

Abstract Data Type addresses an expressive power of concept in programming languages. It facilitates not only the modularization and information hiding, but also performs as an important agent for establishing a large, complex, and manageable program in software engineering. Although many modern programming languages provide different styles of Abstract Data Type constructs, Ada and C++ are two of the most popular programming languages used to teach this aspect at the undergraduate schools in Taiwan. In this paper, we will examine the constructs of Abstract Data Type involved in Ada and C++ and present our experiences on teaching Abstract Data Type in both languages.

1 Introduction

Abstract Data Type (ADT), pioneered by Liskov and Zilles in mid of 70s, is a crucial concept in teaching current programming language[6]. According to the pioneers, there are two essential ideals for a good programming system: *information locality* and *information hiding.* Information locality implies keeping the data together with the operations related to it. The behavior of the real-world object is expressible in terms of the operations and associated data, and the operations are the only ways for accessing this data[4]. This concern leads to the introduction the concept of *modules* into the programming languages during the past two decades. The second essence emphasizes the importance of separating two aspects of abstractions: *specification* (i.e. logical definition) and *physical implementations.* Specification corresponds to the abstractions of the real-world and hides the complex implementation details from the user. It interfaces the abstraction of the real-world with the detailed implementation of the program, thereby preventing the occurrence of undesired side effects[8].

Beginning with SIMULA 67, many programming languages have been providing different level of programming constructs for encapsulating the abstract views into *specification* and *physical implementation.* These encapsulating constructs are used not only to hide the detailed implementation of the ADT, but also to act as a screening device that makes the encapsulated object visible to user (i.e., export or public) or to specify the external information that is made visible within the object (i.e., import). This view of modules as information hiding devices is valuable for establishing more structured, manageable, large and complex programs

Based on a survey conducted by Dr. Richard Reid from Michigan State University (reid@cps.msu.edu), in his newly released report on Oct. 8, 1993 shows that Ada is ranked number two used as the first course language for computer science majors worldwide. Dr. Reid surveys among 365 accademic institutions, the

Table 1: First Language in Computer Science Majors

Type	numbers used	percentile
Pascal	147	40.3 %
Ada	57	15.6 %
Scheme	47	12.9 %
Modula	43	11.8 %
C	28	7.7 %
Fortran	8	2.2 %
C++	8	2.2 %
Modula-2	8	2.2 %
SML	5	1.4 %
Turing	4	1.2 %
ISETL	2	–
Miranda	2	–
Modula-3	1	–
Oberon	1	–
ObjPascal	1	–
Orwell	1	–
Simula	1	–
Smalltalk	1	–
Total	365	100%

top five languages used are: (1) Pascal, (2) Ada, (3) Sceme, (4) Modula and (5) C which dominate 80% of the total usages. There are 57 institutions (around 15.6%) of the total use Ada as their first language, 147 use Pascal language (40.3%), 47 (12.9%) use Scheme, 43 (11.8%) use Modula and 28 (7.7%) use C. The other languages used include Fortran(8), C++(8), Modula-2(8), Modula-3(1), SML(5), Turning(4), ISETL(2), Maranda(2) and others. Table 1 shows the results of the survey.

Among the most popular educational programming languages used in undergraduate school, C++ (i.e., class), Ada (i.e., package) and Modula-2 (i.e., module) are three suitable languages to teach the concept of abstract data type. Here, we will only concentrate on C++ and Ada.

The *class* of C++ includes specifications of data members (i.e., attributes) and member functions (i.e., methods). It can contain both hidden and visible entities. A *private* clause can be used to hide entities and a *public* clause is used for visible entities[7]. In Ada, the use of *packages* and *private data types* also provide the similar facilities in the same aspect[1,12]. The difference from the class mechanism in C++ is that the *package* in ADA is static in semantics. It can not be created or freed dynamically during the run-time. What they provides is to patitionize the program text into manageable units.

Thereby, the mechanism of the Abstract Data Type in C++ and Ada make both languages as the cornerstone of a good programming language which allows us to develop a more reusable and reliable software component in a large and complex environment. In this paper, we will examine programming constructs of ADT involved in both languages and present our experiences on teaching this aspect in both C++ and Ada.

2 Data Abstarction

The paradigm for the early procedural programming languages is to *decide what procedures (or functions) and algorithms to be used*[11]. It emphasises on the *processing* and the computational *algorithms.* Thus, the early conventional languages provide facilities for passing arguments to functions and returning values from functions to support this paradigm.

During past two decades, the size of the program keeps increasing with more complex and larger applications. The process of programs' design has also shifted from the concerning of *procedures* to the *organization of data.* The data with a set of procedures which manipulate it is referred as a *module.* The paradigm for this stage is then change to *decide which modules to be used and partition the program so that data is hidden in modules*[11]. Hence, it is also regarded as the *modular programming* and the notion of data hiding is extended to the notion of *information hiding.* Programming with modules shows to the

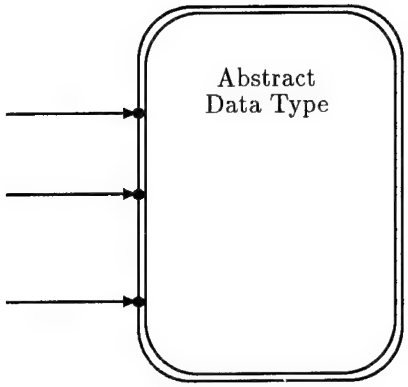


Figure 1: The Interface of Data Abstract Type

centralization of all data of a type under the control of a type manager module. Although the module concept enables *data-hiding principle* but it still does not behave the same way as built-in types. In some applications, this may lead each type-manager define a separate mechanism for creating its "variable" of its type, there is no established norm for assigning object identifiers[11].

To tackle this problem, languages such as Ada (i.e. package) and C++ (i.e. class) allow a user to define types that behave in the same (or nearly the same) way as built-in types. Such types are often called an *Abstract Data Type.* What is the notion of abstraction to an ADT? The specification is an exact statement of the abstract entities which provides an interface with outside world (see Figure 1). Under the concept of ADT, the programming paradigm elaborates to *decide which type need and provide a full set of operations for each type.* There is no need for more than one object of a type, and the data hiding programing style still suffice with using modules [11].

The specification of an ADT is a form of contract between a user and an implementor. The method of constructing programs using specifications is therefore sometimes called programming by contract. With the specification in mind, An user can assume that the actual performance of the ADT will act in accordance with the specification.

The traditional procedural programming languages are long being criticized not provide enough control of visibility. The Ada's *package* and C++'s *class* overcome this by allowing us to place a wall around a group of declarartions and only permit access to those which we intend to be visible[1].

The evolution of programming design beginning from *procedures and functions*, *module programming* to currently well-established *abstract data type.* Barnes[1] lists four major advantages of using abstract data type: (1) simplicity (2) integrity (3) reusability (4) implementation independence. This progress provide not only a better framework for students to learn programming language but also give better flexibility for an instructor to design and assign programming projects to students. We will discuss our experiences in teaching ADT through C++ and Ada in the following sections.

3 ADT in Ada

Package is regarded as the notion of the implementation of ADT in Ada. A *package* is divided into two parts: *specification* and *package body.* The *specification* declares the entities to be visible and is the interfaces to the outside world, the *package body* is the detail implementation of the entities declared in the *specification.*

Abstract data type can be implemented in Ada with packages. Packages offered an information hiding mechanism in Ada and can be divided into two parts: one is the specification of the package and the other is the implementation details of the specification. Basically, the package specification serves as the interface to the user of the package. There can be declarations of types, variables and subprograms in the interface. The implementation can contain anything but already appear in the interface, such as declaration of types, variables, subprograms and most important, their code. The specifications and implemenations can be referred to as the *public* and *private* views respectively.

3.1 Private Types

Abstract data types can be implemented in packages by the private type. It serves to deliver an interface to hide the implementation details of types from the user in the visible part of package. A private type can be in two forms by associating with different attributes, *private* or *limited private*, depending on the semantics needed by the ADT designer. Following is the brief descriptions of the private type.

Operations offered by private type in the package can be passed as parameters, compared for the logical equality and assigned values of other objects of the same private type. That is, the private type restricts the operations of those privated objects to the assignment and logical comparison and so on. Compare to the limited type, it gives the user more freedom in associated with the type. Like the base types in the programming languages such as integer or real, user can make no assumption about the internal representation of the base type, unless the language designer provided one.

3.2 Limited Private Types

Moreover, Ada provided a mechanism to control all operations allowed in an user defined data type, limited private type also known as limited type. Users can do nothing with the limited type without the permission of the designer. Thus, the limited type provides complete control over the operations of a type to implement the notion of *information locality.*

3.3 Examples

In figure 2, Package PriQue implements a priority queue. Note that type *Uptr* is declared as limited private to provide the complete control over the queue. The limited type confined the operations over the resource, Uptr, to the functions declared in the specification of the package PriQue. User now must use *enqueue* to add a element to the queue and remove elements from queue with *dequeue.*

```
  Uptr is access usernode;
  type node;
  type nodeptr is access node;
  type node is record
        prio : float;
        value : Uptr;
        next : nodeptr;
  end record;

Package PriQue is
      type heapsize is private;
      type blockcnt is private;

      procedure enqueue(up: INOUT Uptr; prio: INOUT float);
      procedure enqueue(prio: INOUT float);
      function dequeue return Uptr;
      function emptyque return Integer;
      procedure initqueue;
      function first_value return float;
      function first_ptr return Uptr;
End PriQue
```

Figure 2: Package specification of Ada

Ada offered a strong mechanism in constructing the world of abstract data type. To a student who is innocent in the art of programming, it seems not so appropriate to use Ada as their first programming language. In our teaching experience, the strong typing and versatility of the type system in Ada often confused students. Since they are new to the data types and can make lots careless syntax errors results in the progress of the learning curve slow down. This is a fact that exists and we should be aware of it.

4 ADT in C++

Data abstraction is one of the basis of OO programming, which is the rapidly emerging standard paradigm for programming today. C++ is probably the most widely used object-oriented language today. It provides classes with multiple inheritance and data abstraction using *public, private, and subtype-visible(protected)* declarations. Classes can be parameterized with type parameters. The inheritance hier-

```
#ifndef calendar_h
#define calendar_h
//
// virtual base class Calendar which can
// be customized with all data structures
//
class Calendar{
  public:
      virtual int emptyqueue() = 0;
      virtual void initqueue() = 0;
      virtual void enqueue(void *n, double prio) = 0;
      virtual void *dequeue() = 0;
      virtual int message() = 0;
      virtual void init_calendar() = 0;
      virtual double top_pri() = 0;
      virtual void *top_element() = 0;
};
#endif
```

Figure 3: Virtual Class of Calendar

archy is used for subtyping purpose.

Figure 4 is class Heap2 which is a priority queue and implemented by using 2-heap. Class Heap2 is an ADT implemented with *public, protected, private* as access control mechanisms of attributes and operations. To allow a variety of implementations can be used as priority queue, the C++ provides a *virtual base class* for each implementation to inherite from the same abstract class without changing the platform. Figure 3 is the declaration of a virtual class Calendar in C++. A virtual class can not be instantiated but it provides a generic interface for user to use different types of priority queue implemetation dynamically. Figure 5 shows the hierarchy of virtual class and its inheited class in C++.

Thus, OO model can support a good software engineering design in terms of modularity, abstraction, reduce coupling and enhance cohesion for a program. It also provides a seamless transistion from design to its implementation.

```
//
// The following declarations are used by the
// enqueue and dequeue operations on a priority
// queue using the implicit heap see D. E. Knuth,
// The Art of Computer Programming.
//
#ifndef heap_h
#define heap_h
//
#include "calendar.h"
//
typedef struct Heap_node {
double prio;
void *value;
struct Heap_node *next;
struct Heap_node *dup_next;
} Heap_node;
//
typedef Heap_node *Heap_noderef;
//
//
//
class Heap2: public virtual Calendar {
  private:
      int maxsize,maxheapsize;
      int heapsize_cnt;
      int Dup_flag; // flag for Duplication mode
      int Dup_input_buf_flag;
      int Dup_output_buf_flag;
      int this_is_first_node;
      int this_is_first_dequeue;
  public:
      Heap_noderef *heap ;
      int heapsize ;
      Heap2();
      Heap2(int flag);
      void initqueue();
      int emptyqueue();
      void enqueue(void *n, double prio);
      void *dequeue();
      int message();
      void init_calendar();
      double top_pri();
      void *top_element();
};
#endif
```

Figure 4: 2-Heap as a Priority Queue

5 <u>Conclusion</u>

The essential characteristic of an ADT is the separation between its concept and implementation which is

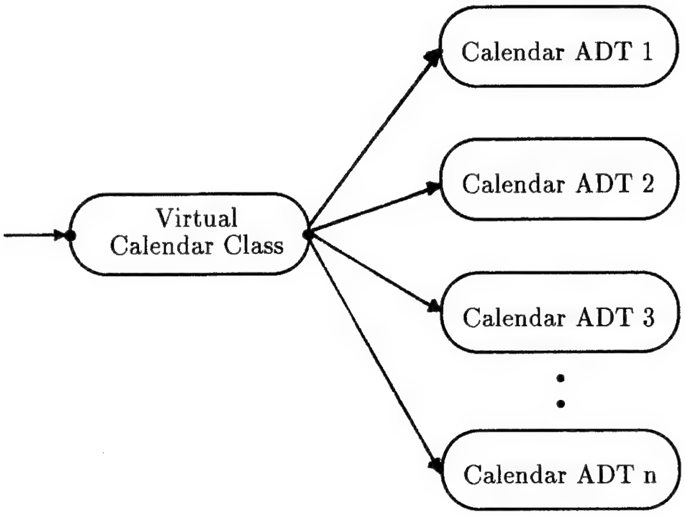


Figure 5: The Virtual Class in C++

Table 2: Feature comparison

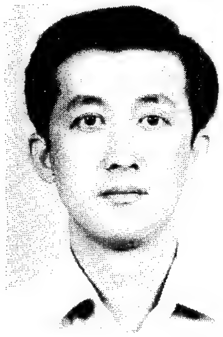
Features	Ada	C++	Modula-3	C
Strong typing	yes	some	yes	no
OO	no	yes	yes	no
generics	yes	no	yes	no
garbage collection	no	no	yes	no
exception	yes	no	yes	no
concurrency	yes	no	yes	no
user define operator	yes	no	yes	no

also used to describe the ability of *information hiding.* The programmer of the data abstraction can give descriptions of the set of values and the set of operations that define the ADT, but the implementation of the type is inaccessible–that is, hidden[9]. Table 2 compares the features among Ada, C++, C and Modula-3.

The Abstract Data Type addresses an expressive power of concept in programming languages. It also facilitates as an important agent for establishing a large, complex, and manageable program in software engineering. From our experiences in teaching ADT with Ada and C++, we notice that both languages provide well-established platforms for learning ADT. The *virtual class*, *inheritance*, *polymorphism* and *protect of access control mechanism* in C++ show its preeminence over Ada, Yet the *limited type*, *concurrency of task*, and *parameterized mechanism (generics)* in Ada show its uniqueness over C++. There is a fact we must confess that since C language have dominated the world of progrmming language for over ten years, teach ADT through C++ have a conventional advantage over ADA. Though ADA is a superior langauge for software engineering and real–time programming and also object–based, students need more time to comprehend the picture of ADT from ADA over C++.

Biography

Dr.Huei-Chung Chu is an associated professor and chairman in the Department of Information Management Science at National Defense Management College. He received his Ph.D. degree in Computer Science from Illinois Institute of Technology in 1990 and M.S. degree of Computer Science from Northwestern University in 1979. Dr. Chu's current research interests include concurrent programming, distributed programming language, distributed Ada and computer security.



Dr. Ke-Hsiung Chung is an associate professor in the Department of Information Management Science at National Defense Management College. He received his Ph.D. degree in Computer Science from Purdue University in 1993 and M.S. degree of Computer Science from Georgia Institute of Techno–logy in 1986. His current research interests include object-oriented technology, object-oriented simulation, simulation methodology, concurrent system, parallel and distributed stochastic simulation.

Mr. Bau-Hong Yuan is an instructor in the Department of Information Management Science at National Defense Management College. He received his M.S. degree in Computer Science from Illinois Institute of Technology in 1990. His current research interests include programming language, concurrent and distributed programming.

Dr. Chu can be contacted in P.O.Box 93–75, Taipei, Taiwan, R.O.C.

Dr. Chung can be contacted in P.O.Box 90046–15, Chung–Ho, Taipei, Taiwan, R.O.C.

Mr. Yuan can be contacted in P.O.Box 90046–15, Chung–Ho, Taipei, Taiwan, R.O.C.

References

[1] J. Barnes. *Programming in Ada.* Addison-Wesley Publishing Company, New York, N.Y., second edition, 1984.

[2] K. H. Chung, J. Sang, and V. Rego. A Performance Comparison of Simulation Calendar Algorithm: An Empirical Approach. *Journal of Software Practice and Experiences*, vol 23(10), pp 1107-1138, Oct. 1993.

[3] K. H. Chung and V. Rego. Sol: An Object-Oriented Platform for Evented-Scheduled Simulation *SCSC'93, The Proceedings of the 1993 Summer Computer Simulation Conference*, pp 972-977, July 19-21, 1993.

[4] C. Ghezzi and M. Jazayeri. *Programming Language Concepts.* John Wiley and Sons, New Yory, N.Y., second edition, 1987.

[5] B. Henderson-Sellers. *A Book of Objected-Oriented Knowledge.* Prentice Hall, New York, N.Y., 1992.

[6] B. Liskov and S. Zilles. Specification techniques for data abstraction. *IEEE Transaction on Software Engineering*, vol 1, 1975.

[7] R. Sebesta. *Concepts of Programming Languages.* The Benjamin/Cummings Publishing Company, Inc., New York, N.Y., second edition, 1992.

[8] R. Sethi. *Programming Languages Concepts and Construct.* Addison-Wesley Publishing Company, New York, N.Y., 1989.

[9] R. Sincovec and R. Wiener. *Data Structures Using Modula-2.* John wiley and Sons, New York, N.Y.,1986.

[10] S. Harbison. *Safe Programming with Modula-3. Dr. Dobb's Journal*, pp 88-95, Oct. 1994.

[11] B. Stroustrup. *The C++ Programming Language.* John wiley and Sons, New York, N.Y., second edition, 1993.

[12] D. Stubbs and N. Webre. *Data Structures with Abstract Data Types and Ada.* PWS-Kent Publishing Company, Boston, MA., 1993.

A Year-Long Sequence in Software Engineering to Use the Ada Language in the Undergraduate Curriculum

William G. McArthur and Rick E. Ruth
Shippensburg University (PA)

Abstract. *Two existing courses were integrated into a coherent sequence in software engineering utilizing Ada. The first course now introduces Ada and re-use of software components in the context of an advanced study of data structuring. In the first course, the students are also introduced to the Microsoft Windows*^{TM} *environment for program development. In the second course, the students study the software development process as members of four person programming teams which are creating a real Windows application in Ada. The Windows programming allows the students to experience software re-use via dynamic link libraries (DLLs) as well as in Ada.*

Introduction

This paper is a not-yet-final report of a project funded by a Defense Advanced Research Projects Agency program for Curriculum Development in Software Engineering and Ada[1]. The project entailed combining two independent, existing courses, *Advanced Data Structures* and *Software Design for Information Systems,* into the sequence of courses, *Advanced Data Structures Using Ada* and *Software Engineering Using Ada.* The sequence is designed as a required upper division capstone experience which will integrate and balance the fundamentals taught early in the program with the current technology topics explored in upper division courses.

The new sequence offers a unique combination of pedagogical approaches. The first course emphasizes the theoretical aspects of large program development in Ada, stressing information hiding, modularity, data encapsulation, and coding and documentation standards. The second course consists of a semester-long realistic project to be completed by groups of four students as they take the project through the software engineering development process. The project is developed using various CASE tools and is implemented as a Microsoft Windows application. This combination of techniques provides a most useful blending of theory and practice for senior computer science majors.

Rationale

Many computer science curricula contain a common core of courses that may end as early as the second year; after which different tracks may be pursued. This is our current model at Shippensburg University. By providing a capstone experience, real integration can occur and the format of a year-long sequence provides the depth needed for this experience. Such preparation would be ideal prior to internship or co-op assignments or summer jobs in the field, but is certainly needed prior to that first real job.

The Ada programming language provides a "real world" environment and is conducive to good software design and engineering. Aside from the intrinsic merit, our location near Letterkenny Army Depot, the Navy Ships Parts Control Center, and the greater Washington D.C. area is an added advantage of using the Ada language. Some of our graduates have been granted interviews and secured that all important first job based, in part, on the limited exposure to Ada that we currently provide.

The project was divided into three phases: *Course Sequence Development, Advanced Data Structures Implementation,* and *Software Engineering Implementation.* We will comment on each phase and conclude with the results from the actual implementation.

Major Considerations

Course Sequence Development involved both of the principal investigators working together and separately on the various tasks of this phase. Course outlines were developed jointly to ensure completeness, continuity

[1] This project was sponsored by: United States Air Force, Air Force Materiel Command, Phillips Laboratory (PL), Kirtland AFB, New Mexico 87117-6008

and, integrity. The following topics, from ACM/IEEE's *Computing Curriculum 1991*[6] where indicated, are covered in the *Advanced Data Structures/Software Engineering* sequence.

AL1

- Review and implementation of "classical" data structures and appropriate operations to include: Arrays, Strings, Lists, Tables, Stacks, Queues, Trees, and Graphs
- Emphasis will be placed on dynamic allocation, recovery, and reuse of storage and linked implementation (also PL6)
- Design of (new) data structures, reuse, space vs. time issues

AL2 and AL6

- Implementation of more modern data structures (ADT's) and appropriate operations to include: Rings, Sets, B-trees, Hashing, Maps, Internal and External Searching and Sorting, and File Structures (OS7)
- Analysis of Algorithms
- Levels of abstraction, modularity, interchange of implementations
- Large program development issues

Other Advanced Data Structures Topics

- Overview of the entire sequence
- Documentation and coding standards
- Advanced introduction to the Ada language
- Data encapsulation and information hiding

SE2

- software life-cycle models
- software design objectives
- documentation
- configuration management and control
- software reliability
- maintenance
- specification and design tools; implementation tools

SE3

- informal requirement specifications
- formal requirement specifications

SE4

- functional/process-oriented design
- bottom-up design; support for reuse
- implementation strategies
- implementation issues; performance; debugging and antibugging

SE5

- structured walkthroughs

Advanced

- testing strategies
- CASE tools
- metrics
- prototyping
- version control
- end-user considerations
- standards and international issues
- reuse
- safety
- reliability
- portability
- project organization
- scheduling

Other Software Engineering Topics

- team formation
- formal reviews
- communicating with users
- progress presentations to management
- PC programming with a graphical user interface
- PC programming with a database

The decisions on documentation and coding standards were also made jointly. The guidelines and instantiations contained in Chapters 2-5 of *Ada Quality and Style: Guidelines for Professional Programmers*[3] were chosen as the basis for coding standards. However, we found that we still needed to add a few of our own examples. The added examples deal almost exclusively with headers--for files,, package specifications, package bodies, program unit specification, and program unit bodies. The documentation standards are an adaptation of industry standards previously developed by one of the authors.

Having defined the philosophy, content, and standards for the courses, a review of possible textbooks was then undertaken. Since the objectives of the first course involve an advanced introduction to the language, a second look at data structures and ADT's with an emphasis on reuse, file structures, an orientation to object-oriented design as well as software engineering methodology, finding a single text seemed unlikely, and in fact, none surfaced. Two or even three texts should be used. One might present a good, but brief, advanced introduction to the language which means that it should not assume any prior knowledge of the Ada language, but it may assume a solid familiarity with some high-level language like C or Pascal. It should also emphasize the added features and not present just a Pascal-like subset and approach to programming. A second text might cover data structures and ADT's emphasizing reuse, and a third might study files (or external data structures). No suitable text has been found for the advanced introduction to the language. Thus, it was taught from class notes. However, we do recommend *Software Components with Ada* by Booch[1] and *File Structures with Ada* by Miller and Petersen[4] to satisfy the other major objectives of the course.

If the objectives for the first course seem somewhat overwhelming, please note that some economy is made by having groups of students working on different "packages" concurrently. Also, emphasis is placed on the design and testing of these packages rather than on applications of any appreciable size.

Applications are the thrust of the second course. The straightforward content and lesser dependence on the language of instruction allow for a broader selection of texts for the second course. Here we recommend *Software Engineering* by Pfleeger[5] which is language independent. The textbook is supplemented by a series of lectures on the program development environment.

The choice of compilers and tools software was limited by a small budget and a situation which dictated a decision too early in the process. Because of the obvious impact of the compiler/programming environment on the second course and the more practical need to access and manipulate data files used by other software, this decision was also made jointly. Academically the selection was further narrowed by decisions to go with a PC based compiler and to allow for Windows programming. The real world projects submitted by actual users and assigned in the second course have of late been exclusively PC based. No change in that regard was anticipated. Also, the continued growth of Windows programming and the need to provide a more graphical user interface (GUI), led us to the PC/Windows environment. Even with the above constraints, we were able to find a reasonably priced compiler and associated software tools to meet these needs.

Our chosen compiler is OpenAda Windows v2.0 from Meridian Software Systems, Inc., 10 Pasteur Street, Irvine, CA 92718. OpenAda Windows, however, must be used in conjunction with either the Microsoft Windows Software Development Kit or Borland C++ 3.1. We used it with the latter since Borland's C++ and Pascal With Objects are already used in other courses. OpenAda Windows easily allows DOS I/O programs to be run as Windows applications. The help file was improved by one of the authors to include the *Guidelines*[3] and more links for easier searching of the L.R.M.[2]. We have chosen two commercial CASE tools: PROTRACS scheduling software from Applied Microsystems and OpenSelect Windows from Meridian. In addition, Microsoft Word for Windows was chosen as the standard word processor for documentation and help file development. We felt comfortable with the choice of a Meridian compiler since we have successfully used Meridian's AdaVantage, one of the first DOS based Ada products in earlier experimental versions of the first course.

One problem with programming in the Windows environment is the complexity of the application program interface (API). We have attempted to reduce some of the complexity by means of a *preprocessor* which allows for simplified specification of certain windowing constructs. At this writing, the preprocessor is a manual one, but it is in the process of being automated. Another important tool is the *librarian*. The librarian utility provides for many of the configuration management needs of a project. The librarian provides versioning, development tracking, and control for documents, files, and code. This tool is (at this writing) being developed in Ada for Windows. A third major piece of the development environment is an interface to xBase files. We are using DataBoss for Windows from Kedwell Software to generate a DLL to provide for the necessary access. For projects with more modest file access needs, we have developed a table package which uses a hashed file as its storage mechanism. For printing reports, the students have available an Ada printing package with a simple interface. Also, the students are provided with a number of source code templates for dealing with the most common Windows constructs. Because of budget constraints, no commercial debugger was available; however, a simple debugging expert system was written by a group of students from our Artificial Intelligence course.

Once these choices were made, the following tasks were accomplished somewhat independently for each course.

- design syllabi
- develop readings
- develop classroom examples and design overhead transparencies
- develop exercises and solutions
- design examinations and solutions
- develop team guidelines
- design team projects

Implementation of ADS

The first course to be converted, *Advanced Data Structures*, was originally taught using the Pascal language prior to the introduction of that language into the beginning courses in our curriculum which were taught using FORTRAN. Once Pascal became the language of instruction for those beginning courses, we introduced the Modula-2 language into the advanced data structures course. Although an improvement at the time, Modula-2 is not an appropriate choice in light of today's wider range of languages from which to choose. About two years ago we began experimenting with using Ada in the course because 1) it lends itself to more object oriented design 2) it maintains a Pascal-like syntax and 3) most importantly, it is a real production language. The only disappointment was the lack of quality, yet reasonably priced, course materials and software.

Phase One of the project provided the means for addressing these concerns, by allowing the time needed and the associated software tools to develop course designs and materials. The implementation of *Advanced Data Structures Using Ada* consisted of preparing course notes and actually teaching the course during the fall of 1993 as designed in Phase One. A brief outline of the way the course was structured is as follows:

- three weeks on an advanced introduction to the language
- four weeks on data structures and ADT's Part I (stacks, lists, strings, queues, and rings)
- three weeks on data structures and ADT's Part II (tables, maps, sets, graphs, and trees)
- four weeks on sorting/searching and file structures

Exams were given after each segment with the last being a not-quite-comprehensive final exam, and ten programming projects were assigned. As mentioned earlier, students worked in groups of four to develop and test these packages. The groups were assigned different projects and during the testing phase were given another group's project to test and critique.

For very practical reasons, the actual instruction of Windows programming, except for the very limited use of some Windows programming templates, was delayed until the second course. Even then, the preprocessor was used to hide many of the gritty details and save precious class time. The first course laid a solid foundation in standard, and quite portable, Ada programming. It presented the "nuts and bolts" (so to speak) while building as large a base to stand on as time would possibly allow. For the sake of completeness and portability, a standard text-based user interface was employed. Besides, we knew there would be students in the second course who had already taken the earlier, more experimental version of the first course which definitely didn't include any Windows programming or involve a graphical user interface.

Implementation of SE

The software engineering course had evolved over a number of years. The genesis for the course was a rather unstructured projects course for seniors. To that course were added life-cycle models, documentation standards, peer reviews, and some CASE tools. The course had been a popular one with the students and with industry recruiters. The implementation environment was unrestricted, but tended to standardize on a PC network running dBase IV or FoxPro. As we moved to a new paradigm for the course, we wanted to maintain all of the good aspects of the old course.

The new implementation environment is proscribed rather than unrestricted. The disadvantage is lack of flexibility, especially for prospective users. The main advantage is that of uniformity, with the possibility of teaching units on aspects of the environment. The new environment consists of a standalone PC or network running Microsoft Windows 3.1 or later. The programming language is Ada (for Windows) supported by Borland C++ for Windows. The Borland Resource Workshop is used for development of icons, menus, dialogs, and other Windows resources.

Prior to the beginning of the semester, potential users are contacted to gather possible projects. Acceptable projects must be of appropriate size (about 560 staff hours to complete) and fit the constraints of the implementation environment. Potential users are told that they must be willing to devote some time at the beginning of the project to be interviewed by a development team. The user must supply enough information about the project to fill out a *Software Development Request* form (about a half page). The first day of class, the students fill out a *Team Formation Questionnaire* in which they provide some information that helps to determine the development groups. Besides stating their experience, the students can indicate interest in a particular project or list students that they would and would not like to work with. On the second day of class the students are assigned to four person development teams. Each team has a different assigned project and each team is assigned another team's project which they are responsible for reviewing and testing.

The teams are organized into roles: leader, librarian, test engineer, and documentation editor. The roles must be rotated as project phases change so that each student gets some experience in each role. The project passes through the phases (the numbers indicate team role rotation): preliminary specification(1), requirements specification(2), design specification(3), user's guide(1), coding and unit testing(4), . system integration(1), acceptance testing(2), and delivery(1). Each of the documents (and the code) are subject to formal peer review and to review by the professor. The students are given a comprehensive set of guidelines and forms for reviews. During the design specification phase, each team prepares the acceptance test for its co-project. The acceptance test is reviewed by the development team. The acceptance testing is accomplished for the co-project.

All of this activity happens during a 14 week semester, so there is a brisk pace of activity. Each student is required to maintain a *personal dairy* containing a running total of hours. The teams are required to maintain *meeting minutes* and to submit a *weekly* *progress report* (on Fridays). During the project, the team makes two formal oral presentations. At the conclusion of the requirements specification, the team must prepare and deliver a ten minute presentation to "upper management" to "sell" the worth of the project and to "sell" the worth of the team. The presentation is of a non-technical nature. At the midway point of coding, the team must present a ten minute progress report to the "project manager." This second presentation is technical in content. The students are encouraged to make good use of audio-visual material for the presentations.

A bit less than half of the class periods are used for lectures. Some of the class periods are used for peer review meetings. The remaining class periods are used for team meetings. The students are warned that each hour that they meet together, time is being expended at a four-fold rate, so the time must be used wisely. The professor's role changes among *teacher, manager,* and *programming expert* as the context requires.

At the close of each phase of development, each student fills out a confidential *Group Evaluation* in which they answer a number of questions about team organization and efficiency. Student class attendance is strictly enforced, since many class periods are used for team meetings. Each development team formally presents and demonstrates their project during final examination week. The final presentation usually lasts about an hour. At that time, the students submit a last group evaluation, in which they are asked to grade themselves and the others for both quality and quantity of work. The student diaries are also turned in at this time. It usually takes about 15 minutes following the group presentation to come to a tentative set of grades for the students. The students are then consulted and a grade for the course is negotiated.

A major side-effect of the course is that the students write hundreds of pages of documentation that is drafted, revised, reviewed, and revised some more. The students are encouraged (read as required) to use the Microsoft Word for Windows word processor, its spell checker, and its grammar checker. Standard outlines are provided for all documentation as Word for Windows documents.

The implementation of *Software Engineering Using Ada* also consisted of preparing course notes and actually teaching the course during the spring of 1994 as designed. The course was improved by using the Ada language and associated tools to do Windows programming. The level of sophistication of the projects was elevated by the student's knowledge from the first course in the sequence.

Results

This year-long project resulted in the strengthening of our curriculum which will now provide a more integrative, cohesive, and (hopefully) real world experience for our students and others. A complete package of educational materials suitable for adoption by educators planning similar course sequences will be available sometime after the completion of the project. It will consist of two volumes, one for each course, containing descriptions of the project, the materials mentioned above, and final reports.

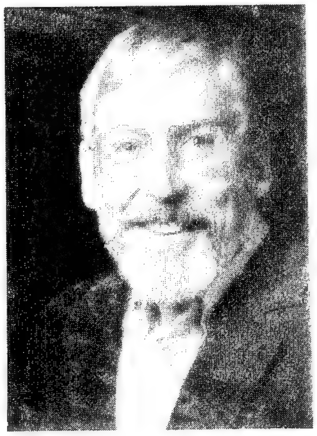
Bibliography

1. Booch, Grady, *Software Components with Ada: Structures, Tools, and Subsystems,* Benjamin/Cummings, Menlo Park, CA 1987.
2. Ichbiah, J.D. et al, *Reference Manual for the Ada Programming Language (ANSI/MIL-STD-1815A-1983),* U.S. Department of Defense, 1983.
3. Johnson, Kent, et. al., *Ada Quality and Style: Guidelines for Professional Programmers,* Third Edition, Software Productivity Consortium, Herndon, Virginia, 1992.
4. Miller, Nancy E. and Petersen, Charles G., *File Structures with Ada,* Benjamin/Cummings, Menlo Park, CA, 1990.
5. Pfleeger, Shari, *Software Engineering (2nd Edition)*, Macmillan, New York, 1991.
6. Tucker, Allen B., et. al., *Computing Curricula 1991: Report of the ACM/IEEE-CS Joint Curriculum Task Force.* 1990.

The Authors

William G. McArthur (wgm@ark.ship.edu) was graduated from the Pennsylvania State University in 1969 with a Ph.D. in mathematics. He came to Shippensburg University following graduation in March of 1969. In 1970 he began to migrate toward the field of computer science. The change of fields was completed by means of a sabbatical leave in the computer science department of Duke University in 1977. Since then he has mainly taught computer science courses at the university. He helped to develop several of the courses in the computer science curriculum, including Programming Languages, Compiler Design, Artificial Intelligence, and Software Design for Information Systems. He was the first recipient of the annual Faculty Research Award at the university in 1977. Also in 1977, he earned the Certificate in Computer Programming (CCP) from the Institute for Certification of Computer Professionals as one of the first thirteen people to be certified in scientific programming. He was elected to Sigma Xi Scientific Research Society in 1979. He has been elected for four terms to the Board of School Directors of the Shippensburg Area School District and has served two terms as the president of the School Board. He has authored several papers and given several invited addresses. He has co-authored three computer science textbooks published by Prentice-Hall. Since 1983, he has been a consultant to the computer industry.

Rick E. Ruth (rer@ark.ship.edu) was graduated from Ohio University in 1982 with a Ph.D. in mathematics and additional graduate work in computer science. He accepted an appointment to Shippensburg University as an assistant professor of mathematics and computer science and has taught primarily computer science courses since that time. For the past four years he has taught the Advanced Data Structures course migrating it first from Pascal to Modula-2 and then very recently from that language to Ada. His involvement in curriculum matters ranges from encouraging and supporting the use of Pascal in the introductory courses to participating in the restructuring of the undergraduate and graduate curricula to helping gain approval of and implementing a new university-wide general education program. He has served on both departmental curriculum committees and has also chaired the university-wide curriculum committee. He has published in his field and has refereed and reviewed for publication papers and books in mathematics and computer science. He served for three years as assistant department chair. His primary responsibility was undergraduate programs of which there are six -- three in mathematics and three in computer science. He also directed the department's microcomputer laboratory, which houses 25 IBM compatible machines networked via ethernet and running Novell software. He has presided over the university's faculty association and is currently serving a two-year term as associate provost of the university.



A COMPREHENSIVE SOFTWARE ENGINEERING EDUCATIONAL EXPERIENCE

George C. Harrison

Norfolk State University
Norfolk Virginia

Summary

Between July 1, 1992, and August 15, 1993, we tested some methodologies on both software engineering and Ada education. Funding for these investigations came directly from the Advanced Projects Research Agency (grant number MDA972-92-J-1024). The program had two primary foci. Students were to learn and experience software engineering without concern for the implementation, testing, and maintenance. Other students were to learn Ada in about eight weeks and implement the software engineering project in the remaining time. The first goal was a success by almost any measure. The second did not meet expectations. We will describe the experiences of the students, the project assistants, and the instructor in trying to test these methods of teaching and learning.

Teaching and Learning

For eight years the author taught software engineering and Ada courses to juniors. The Ada course seems to accomplish the general objectives of learning to use and apply the language to general, systems, and, some real-time programming. That course presupposed the students' successful completion of a Pascal-based, traditional CS2 syllabus. Students would learn all the material in Barnes.[1] The emphasis would be on using generic packages for abstract data types (about 3 weeks) and practicing with tasks for problems in concurrent programming and other applications. Both teacher expectations and student satisfaction levels and aptitude were gratifying.

Software engineering teaching and learning experiences were quite different. Students always evaluated the course during the final examination period after the grades were computed. They also suggested alternative methodologies for teaching, learning and experiencing this subject.

The impressions shared by students and the instructor were not surprising. There was not enough time to fully complete both the detailed design document and the implementation. Team coordination and management were lacking; there was either a lack of leadership or members competed for control.

Old Methods

We tried many methods.

1) The most common procedure appears to be to present a problem statement to the students. Divide them into teams. Allow the teams to self-organize. Lecture about 2 hours a week allowing the remaining time for team interaction with the instructor and the rest of the class. This method does allow for some helpful intrateam communications.

2) The second method is similar to the first, but here the instructor presents a different problem statement to each team. This does not allow for much team to team communication, but it does appear to unify each team quicker.

3) We also attempted to use the method of Tomayko.[2] The entire class acted as a single software development team with specific individuals or subgroups assigned to one or more duties: Administrator, Auditor, Librarian, Requirements Group, Design Group, Test Group, Implementation Group. This organizational procedure did have some significant successes, but there were two major flaws. No one student experienced all of the life cycle tasks. A delay with one student or a subgroup held up the entire class project.

4) In terms of student satisfaction, the model of doing only maintenance was most successful. In this case, students received a user's guide, design, and code for a graphics oriented game. They created a significant upgrade and worked on making the documents consistent with the new version. The only real problem was that they were not creating new documents nor making any real data flow or structural decisions.

5) The final method attempted is similar to the second and the third. Students were to divide into subgroups to write modules for later integration. Students would work as a class team on integrating information into documents. This procedure is extremely difficult for an instructor to manage. It has all the problems of methods 2) and 3); also the weakest team slows the entire process.

Ideals and Student Profiles

There are only fifteen weeks in the semester. Ideally, we would like to have a motivated group of junior computer science students. We would provide them with experiences in the entire software development life cycle -- at least through the implementation of the project. We would furnish some CASE (Computer Aided Software Engineering) technology to provide a smooth transition between requirements and design stages (Charts, Data Flow Diagrams, Data Dictionary, and other tools).

Using Ada as a second procedural language for the implementation would be best for a successful project. It would also reinforce the abstract data types that students experienced in their previous CS2 course.

However, the student profile, at least at Norfolk State University, is somewhat unusual. Many come from families who live well below the poverty level. Many must work for self support. Only about 20% of junior computer science students own their own computers or have access to one outside the University environment. Class attendance, team meeting, and laboratory hours often must fit their work schedule and family responsibilities.

"The Best Laid Plans..." Syndrome I

There is also a problem for the instructor that is probably common throughout the country. It seems that there is no way to predict the degree of success of student experiences. Generally, the instructor will not know the number of students nor their capabilities before the first class meeting. Initial team organizations may appear to be potentially successful; students first experience the sharing of ideas and start learning about depending on others to integrate ideas into a common formulation. There are the apparently inevitable problems with what the author calls the 'Etc. Team.' This team formed from the students left out of other teams needs much attention. These students may be below average or have characteristics that do not make them worthy 'team players.'

A Different Model

How could we give a first rate software engineering course while guaranteeing quality experiences and some expertise in using Ada as the implementation tool? As mentioned before, software engineering students evaluated the course and gave suggestions for improvements. One comment, repeated through the years, was to make the course language-independent and not require the implementation.

The instructor took this suggestion seriously. Could a course be designed and implemented to give comprehensive coverage? The syllabus would include the creation of a user's guide, requirements definition document, and a detailed design document. What can be done to teach Ada and finish the implementation? Our experiences proved that we can teach Ada effectively in less than a semester to students who have experienced a traditional CS2 course. However, would there be enough time to implement the software engineering project, if the Ada course followed that course?

We carefully designed two courses to run in sequence with the software engineering course first. Some of the students in this course would already have had Ada. Some students in the Ada course may have already taken software engineering. Thus, the software engineering and Ada courses would not be prerequisites for one another.

The basic syllabus follows:

CSC 380 Software Engineering
Text: Software Engineering, Sigwart, Van Meer, and Hansen: Franklin, Beedle & Associates, 1990

Course Outline:

- **An Introduction to Software Engineering and Life Cycle Models**
- **The Requirements Specifications**
- **System Models**
- **Design and Planning**
- **User Interfaces and Human Factors**
- **Detailed Design**
- **Implementation and Testing**
- **Maintenance**
- **Software Project Management**
- **Software Tools and Environments**
- **Quality Assurance**
- **Societal Issues**

Grading:
2 tests 40%
Team Work
Requirements Specification 15%
User's Manuals 10%
Design Document 25%
Project Legacy (Individual Grade) 10%

The students received the project assignment at the beginning of the semester. They were to completely design a PC-based Student Software Repository (SSR). The final software product was to be for students to keep assignments, examples, and other code in a data base that included annotations for each source code unit in the SSR. The student user would then be able to add and delete items, search for keywords, and retrieve, view, print, and include software in external libraries. The SSR project was to be as fault-tolerant as possible. The user interface was to be designed as a keyboard (non-mouse) search system. The SSR was to run under MS-DOS 5.0 and be written in the Meridian Ada 386 language environment. Common Ada coding standards were to be used.

We selected Meridian's OpenSelect CASE tool to assist students in documentation and the requirements-to-design transition.

Students enrolled in both CSC 301 (Ada programming I) and CSC 302 (Ada programming II) were charged to implement the project. CSC 301 students would learn Ada syntax and rationale by way of the NSITE-Ada CAI (Computer Aided Instruction) package and their text book[1] within 8 weeks of the beginning of the semester. The CSC 302 students, who were experienced Ada programmers would build their Ada programming skills. They would become familiar with the special PC interfaces on Meridian's compiler and then work on teams with the CSC 301 students to assist them in the implementation.

The CSC 301 students were to complete as much of NSITE as possible before the deadline. The grade on the NSITE portion would be 50% of the final grade computed as

(LESSONS COMPLETED)/17 * 50%

They would also be graded on four programming assignments, a final examination (taken after 8 weeks) and their work on the implementation of the Software Engineering project.

"The Best Laid Plans..." Syndrome II

By the fall semester 1992 we were ready to begin our Software Engineering course. We had just begun to transition from a minicomputer environment for programming to a PC-based networked 486 environment. Although the PC's were in place, workers did not install the network until March of 1993. Internal purchasing problems, state regulations, and some confusion during the Verdix - Meridian merger held up our procurement of Meridian's OpenSelect, NSITE-Ada, and the Ada compiler. We did not acquire the OpenSelect tool until the middle of the fall semester.

At that time the documentation for OpenSelect appeared to be based primarily on information processing applications. It did not have significant learning support for the data dictionary, data flow diagrams and Constintine charts.

Despite these problems the four teams worked hard to develop a preliminary and final user's manual, the requirements document, and the detailed design document. The instructor was able to work closely with teams to assist them with the now inevitable internal problems. One team (the 'Etc. Team,' above) did not succeed in meeting minimal course requirements.. One team handed in acceptable but not exceptional work. The others submitted work that was much better than any teams completed in previous years. The answers to test questions reflected a solid base of knowledge and experiences that was far above those in preceding courses. Student satisfaction with the courses was also much higher.

During this and the next (Ada) semester, we had two experienced assistants in the laboratories assisting in giving students help on basic problems. Coincidentally, they were also students in the courses described above.

"The Best Laid Plans..." Syndrome III

Although the Software Engineering course was a success by the instructor's standards, the Ada course was not. The relying on the CAI package and student self-motivation for learning Ada was a misjudgement. The instruction software had some drawbacks and some excellent qualities. Most students easily completed all seventeen lessons. Although most students did well on the testing and programming assignments, many reported that they did not feel like competent Ada programmers by the end of the semester. Two teams out of four managed to produce a working prototype of the software engineering course's project.

As in some software engineering educational models, students worked and learned more in team meetings that in class; this Ada programming course had many of the same characteristics. The instructor felt that he did not have majority control over the day to day learning and working processes.

Two students and the instructor finished the project during the following summer session. We have not released the software to the general public: it only works with Meridian Software System's RAMP command. We working on a version that will remove this dependency and all other implementation-dependent features.

Conclusions

The model used for Software Engineering seems to work well. The characteristics of comprehensive experiences in requirements and design and of using some CASE tools appear to give a solid base for an understanding of software development. It also allows the instructor to give more attention to the teams to assist them in working out complex details and to moderate and detect potential problems. During the fall 1993 semester another instructor used this method of teaching software engineering with minor alterations with comparable success.

The only real disadvantage was totally unexpected by the instructor but noticed by the students: the software engineering teams had little motivation to work towards an implementation. Their goal was to complete a detailed design. This made the design-to-implementation process more difficult for the Ada students. The Ada class used the best of the four designs. Yet, teams implemented the project primarily by way of the requirements and the user interface design rather than any algorithmic development in the design document.

The Ada teaching model that included primary learning by CAI, some testing, and working in teams on a large project was not as successful as expected. Student confidence and satisfaction were low. Apparently, the fact that only 4 students registered for the course during this spring 1994 reflects the dissatisfaction from the twenty or so students who took the course the previous year.

Several benefits overshadowed many of the negative aspects of the project.

The Student Software Repository in its final version should be a useful tool to assist students in the practice of reuse.

The detailed experiences of software definition and design worked better for our software engineering students than more traditional approaches.

Class-based CAI instruction is not the best tool for primary Ada instruction. However, as a supplement or guide for well motivated independent study students, it appears to be quite helpful.

References

1. Barnes, J. P., Introduction to Ada, Third Edition, Addison Wesley, 1989
2. Tomayko, James E., Teaching a Project-Intensive Introduction to Software Engineering, Special Report SEI-87-SR-1, 1987

Biographical Sketch

George C. Harrison is a Professor of Computer Science at Norfolk State University where he has worked for twenty years. He took a Ph.D. from the University of Virginia in 1973 and an M.S. in computer science from Old Dominion University in 1986.

Address:
Norfolk State University
2401 Corprew Avenue
Norfolk VA 23514
g_harrison@vger.nsu.edu

AN UNDERGRADUATE SOFTWARE ENGINEERING COURSE SEQUENCE USING ADA*

Roger Y. Lee and **James P. Kelsh**
Department of Computer Science
Central Michigan University
Mount Pleasant, MI 48859
(517) 774-3774

Abstract

Our department offers one software engineering course in our undergraduate computer science curriculum. In this course we have been trying to teach both the principles of software engineering and software development in an "industrial" environment in a fifteen-week semester. So far, this approach has not worked well, ending up with trivial projects mainly because of time constraints.

Like many other colleges and universities we are trying to address this issue by establishing a two-course software engineering program using Ada. In order for us to provide our students with up-to-data software engineering techniques and an "industry-based" software development environment we propose that this program be composed of two separate courses: Software Engineering and Software Engineering Laboratory.

The main purpose of our undergraduate software engineering course sequence using Ada is to produce qualified software engineers who can work productively on industrial software projects.

I. Introduction

Software Engineering requires the integration of technical skill and managerial ability to analyze, design, implement and maintain reliable, cost-effective software systems that meet the requirements of the user. These software systems include both stand-alone software products and software that is an integral part of a larger computer-based product.

Software systems are becoming larger and more sophisticated. Industry is concerned that colleges and universities are not producing students who can work productively on industrial software projects [2]. Now, we notice that an increasing number of computer science departments offering software engineering are adopting an "industry-based" approach that attempts to provide a real world software development experience within an academic environment.

We have been trying to teach both the principles of software engineering and close

* This project was supported by DARPA SFRC# F29601-93-k-0134.

simulation of the software development process in an "industrial" setting [1] in a 15 week semester. This approach has not been well received by the students so far. When many students have reported that there is too much work to do in one semester, we decided that it should have been a two-semester course.

II. Background

Fortunately, the two-course software engineering curriculum proposal which we sent to DARPA was funded in 1993. With the grant we are developing a two-course software engineering sequence using Ada. This new program will be implemented in the Fall of 1994. Generally curricular matters seem to be receiving greater scrutiny from various bodies in the University. There seems to be a greater reluctance to establish a new program. However, a DARPA curriculum development grant helped our case.

This new software engineering program was designed to introduce the premise that the quality of a software product is governed by the quality of the software process that is used to develop and maintain it and that these processes can be controlled, measured and improved. To gain the knowledge of these kinds of processes and to be familiar with the processes, Computer-Aided Software Engineering (CASE) technology should be incorporated into the program.

III. New Program Structure

The new software engineering program at Central Michigan University will consist of two separate courses. In the first-semester course (Software Engineering) students will learn the principles of software engineering and will be given programming assignments dealing with modularization, information hiding, separate compilation, and data and procedural abstraction using Ada. Students in this course are required to use CASE tools for completing their assignments. The programming project component of this course will be an individual effort.

In the second-semester course (Software Engineering Laboratory) students will learn advanced topics in Software Engineering and design and develop a large software system in groups of students using modern software engineering techniques and CASE tools. Students in this course will have a real world experience in group dynamics, group techniques, communication skills, planning, reporting, reviewing and documentation. A detailed description of each course is provided in the appendix.

IV. Objectives and Description of the Courses

1. Software Engineering

Prerequisite: Data Structures, File Manipulation Techniques, and Calculus.

Objectives: The main objective of this course is to introduce modern software engineering techniques and practices, including Computer-Aided Software Engineering (CASE) and Object-Oriented technologies.

The general objectives of this course are that the students will:

. Become knowledgeable about the steps and stages necessary for the analysis, design and implementation of a software system using software engineering techniques and CASE tools.

. Become knowledgeable about the software engineering process, software project organization and management, software project economics, software quality assurance, and software configuration management.

. Become knowledgeable about using the software engineering methods and tools applicable in the respective application domain.

. Become familiar with programming in Ada.

Description of the Course: This course introduces the principles of software engineering. The emphasis is on methods for requirements analysis and specification, and architectural design of systems.

Topics include the following:

. Software Project Planning
. Software Requirements Analysis
. Software Requirements Specification
. Software Design Fundamental Principles, methods, and representation
. Implementation Considerations
. Testing
. Software Maintenance Concepts
. The Role of Automation
. Ada Features

Some programming assignments using Ada will be given. The assignment will deal with software engineering methodologies. In this way the students will have the understanding of software engineering concepts as well as software engineering techniques, including CASE tools.

2. Software Engineering Laboratory

Prerequisite: Software Engineering.

Objectives: The main objective of the second course (Software Engineering Laboratory) is to emphasize the study and implementation of the software life cycle and closely simulate an "industrial" environment in developing a large software system using Ada and to become familiar with the software process and its management and programming in Ada.

The general objectives of this course are that the students will:

. Become familiar with using software engineering methods and CASE tools applicable in the respective application domain.

. Become familiar with the software engineering process, software project organization and management, software project economics, software quality assurance, and software configuration management.

. Learn to function in an "industry-oriented" environment using the principles of team cooperation and project management appropriate to a company.

. Become familiar with software maintenance, "programming-in-the-large", and programming in Ada.

Notice that course 1 (Software Engineering) intends that the students become "knowledgeable about" the software engineering process, while course 2 (Software Engineering Laboratory) intends that the students become "familiar with " these concepts.

Description of the Course: This course covers advanced topics in Software Engineering such as object-oriented and CASE technologies, software project organization and management issues. The emphasis is on the design and development of a large software system in an "industrial" environment using CASE tools.

Topics included in this course are as follows:

. Software Project Organization and Management Issues
. Software Project Economics
. Software Configuration Management
. Software Risk Management
. Software Quality Assurance
. Comparison of Design Methodologies
. OOA/OOD/OOP/OODBMS
. CASE Tools

A semester-long course project is to create a large software system in a group environment from the software requirements analysis phase to the final report and the final software product delivery with documentation. The project will be implemented in Ada using CASE tools.

The class will be divided into smaller groups and each group will be responsible for analyzing, designing, coding, and testing a subsystem for the intended system. After their unit testing has been done successfully, each group will work on the interfaces between them to make a coherent system. The integrated system testing follows. The students will have some experience in software maintenance by working on each other's group project after the final product has been delivered.

V.Projects and Exercises in the Program

1. Software Engineering

The laboratory exercises for this course include the following:

Lab 1 (CASE Tools): Draw simple structure charts, data flow diagrams, presentation graphs, and entity-relationship diagrams using CASE tools.

At this point, we discuss what CASE tools are and how to use them.

Lab 2 (Beginning Ada): Write, compile, test, and debug an Ada program to interactively prompt the user for values, compute some simple expression, and display the result.

This merely introduces the student to the mechanics of running the compiler and elementary syntax features of Ada. At this stage we discuss Ada attributes, so the student may perceive Ada as a "better Pascal."

Lab 3 (Packages): Write a program using a separately compiled package (typically a stack or queue).

This introduces separate compilation, package specification and body, and allows discussing information hiding. At this point, we hope that the student will see some of the features making Ada suitable for software engineering. Using packages to implement data structures helps to reinforce modularity and data abstraction.

Lab 4 (Exceptions): Write a program using a separately compiled package that announces errors by raising exceptions.

This reinforces the student's appreciation for procedural abstraction and information hiding.

Lab 5 (Generics): Write a program to implement some basic data structure (stacks and queues) storing a generic data type.

Most students appreciate the increased opportunity for both data abstraction and procedural abstrction with generics. Many are pleased with the thought of the reusable code possible with generic packages. Presently, our department teaches Ada only in a 1-hour elective course.

Our challenge in the new software engineering course will include raising the sights of those who already know a little about Ada and introducing the language to those who have not taken our one-hour elective. We plan to devote two weeks at the beginning of the term to Ada programming. Although this is much faster than the one-hour elective course, we hope that the students' maturity and motivation will help the material be absorbed faster. Also, the instructor will not have to explain the significance of Ada features for software engineering, since that will arise naturally as software engineering concepts are introduced.

2. Software Engineering Laboratory

A course project is to design and develop a large software system in a group environment using CASE tools. The implementation language is Ada. The project is divided into the following phases:

Phase I (Team Composition): The class will be divided into smaller groups (normally 4-5 people). Project manager and team leaders will be selected.

Phase II (Requirements Specification): Produce a requirements specification document including the following:

- System Overview
- External Interfaces and Data Flow
- Functional Description
- Performance Description
- Exception Handling

- Acceptance Criteria
- Appendix: The user's view (DFDs; PERT chart; staff allocation chart; walkthroughs, etc.)
- Bibliography

The requirements specification is primarily concerned with functional and performance aspects of a software product, and emphasis is placed on specifying those product characteristics without implying how the product will provide those chracteristics.

Phase III (System Design): Produce a design document including the following:

- External Design Specifications
- Architectural Design Specifications
- Detailed Design Specifications
- Test Plan
- Appendix: The software engineer's view(DFDs; PERT chart; staff allocation chart; walkthroughs; structure chart, etc.)
- Bibliography

The "how to" of product implementation is the topic of software design. In this phase, a design may not be precise or even practical. However, during this stage we can insure that all needs of the customer are met and that the pieces of the design fit together properly.

Phase IV (Programming): The following tasks are involved in this phase:

Pre-coding checklist:

- Review the design
- Read the SRS
- Know the OS, packaged programs, CASE tools available

Coding steps:

- Plan the integration
- Design the module
- Walk through the module design
- Plan how to test the module
- Code each module
- Test the module
- Test the lowest levels of integration
- Get started on the user documentation

Submit the following documents:

- Module design walkthrough
- User's manual
- Program listings (tested, error-free)
- System integration plan

Phase V (System Integration and Testing): Perform the following tasks:

- Function testing
- Performance testing
- Stress testing
- Acceptance testing

Submit test analysis report.

Phase VI (Product Demos & Delivery): After the successful system integration, the students will show in class how the final product works.

The successful transfer of ownership from developer to user involves training and documentation. The final project document should contain the following:

- Cover sheet
- Table of Contents
- Overview
- SRS
- SD

. Program Listings, module design,walk through, and system integration plan

. Test analysis report

. Summary and conclusion

. Appendix:

- Structure charts
- Detailed DFDs
- PERT charts
- Program listings
- Data files
- Sample run
- User's manual
- Test anlasis report
- System integration plan

. Bibliography

VI. Depth-first versus Breadth-first

Anyone who teaches a one-semester course wishes for an opportunity to expand it to two semesters. When the opportunity arises, someone must choose between depth-first and breadth-first exploration. The depth-first approach spreads the course material by expanding the coverage of each topic as it arises. This allows intermingling theory and practice topic by topic. The breadth-first approach seeks to cover all basic topics in the first semester (as in the original course) and uses the second course to pursue selected topics or laboratory exercises.

The pros and cons can be summarized briefly:

Advantages:

Depth-first: The student masters a concept before dropping it for another.

Breadth-first: The student sees all concepts early and understands the interrelationships of topics.

Disadvantages:

Depth-first: Exercises may seem artificial until students have seen the other topics.

Breadth-first: Most of the students' understanding is likely to be superficial for 70-80% of the course.

The educational philosophy may produce a standoff, but we choose a breadth-first approach for pragmatic reason: Our department's major requires a choice from a group of courses including one semester of Software Engineering. This has proven a popular choice for our students, but few elective choices remain. If we offered a depth-first sequence, many of our grduating students would be exposed to only half the topics of a Software Engineering course. Thus, our first course in Software Engineering will be largely unchanged, giving an overview of the topics to those who take only one semester. Our second course is largely a laboratory course, practicing the software engineering techniques taught in the first course.

VII. Summary and Conclusion

With the advancement of computer hardware, the software systems that drive the new innovations are becoming larger and more complex. As the complexity of the software systems increases, the problems associated with developing and maintaining the software increase. Software Engineering is an approach that helps us to design and develop manageable, understandable programs for complex solutions [3]. We must provide our students with modern software engineering techniques, appropriate tools and an adequate environment so that they can design and develop software that is reliable, more efficient to produce, easier to understand, and easier to maintain.

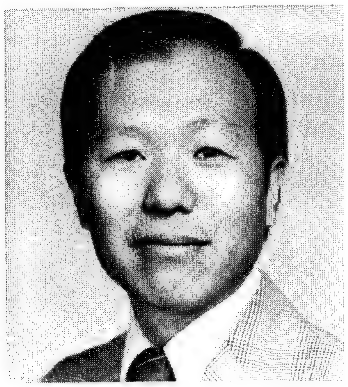
We believe that the proposed two-course software engineering program will help our students to gain not only the necessary software engineering skill but also to have the benefits of a real world software development experience of some sort in an academic environment. Then, our graduates will be ready to work productively on industrial software projects.

References

[1] Clifton, J., An Industry Approach to the Software Engineering Course, SIGCSE Bulletin, Vol. 23, No. 1, Mar 1991, p.296-299.

[2] Ford, G., & Gibbs, N. A Master of Software Engineering Curriculum Recommendation, IEEE Computer, Sept. 1989, P.59-71.

[3] Miles, G. One Approach for Teaching Software Engineering Across the Undergraduate Computer Science Curriculum, Computer Science Education, Vol. 1, No. 1, 1988, p. 53-62.



Dr. Roger Y. Lee is Associate Professor in the Department of Computer Science at Central Michigan University. His research interests include software engineering, compiler construction, programming languages, and computer science education. Dr. Lee is a member of the Association for Computing Machinery, and the IEEE Computer Society.

Dr. James P. Kelsh is Assistant Professor in the Department of Computer Science at Central Michigan Univeristy. His research interests include software engineering, programming languages, computer science education. Dr. Kelsh is a member of the Association for Computing Machinery.

Ada : the Leitmotif of Software Engineering Education

Dr James K Blundell, Associate Professor.
Computer Science Telecommunications Program
University of Missouri--Kansas City
Kansas City, MO 64110
blundell@cstp.umkc.edu

SUMMARY

A fundamental need in computer science education is to provide a coherent and orderly transition for the aspiring student from the realm of a novice programmer to a functional software engineer. A point of heated debate in academic circles is the choice and suitability of the language, or languages, to be used as the vehicle, by which the student will become software engineering literate. This paper will advocate the use of Ada as the programming language of choice for an entire sequence of five software engineering courses, from an introductory programming class to a graduate level software engineering course. The vertical integration of Ada through a course sequence aptly demonstrates the language's suitability as a comprehensive educational tool.

PROGRAMMING AND PROBLEM SOLVING

The educational process that relates to the programming maturity of college students involves more than teaching syntax. (Kushan [1994])The design of algorithms and the solution of software problems are critical components in the pedagogical process and courses are generally designed to combine problem solving skills and programming skills.
Such courses stress concepts relating to

- syntactic knowledge
- semantic knowledge
- structured programming
- internal documentation
- design tools
- debugging techniques

and • product testing

Critical is the sequence in which these concepts are revealed to the student. A critical factor is to reduce the degree of "un-learning" that must take place. A pivotal concept is structured programming, which, according to Lockard [1986], consists of three components

- top-down design
- modularity

and • use of three logical constructs.

The three logical constructs are sequence, selection and repetition.

Structured programming transforms programming from an art to a teachable science. With a structured programming approach, code can be manufactured as opposed to evolving in an idiosyncratic manner.

A "typical" computer science curriculum introduces the student to the concepts of algorithmic analysis using an imperative language, such as Pascal. Once the student has been exposed to "the programming tricks of the trade," the educational emphasis shifts to modularity and abstraction. This transition usually begins with a "data structures" course where the emphasis shifts from "a" programming solution to "an elegant" programming solution. Reusability of code is introduced and this involves organizational issues pertaining to program style, internal documentation and program libraries.

Once the student can generate coherent program modules, then the process to develop "industrial strength" software can be revealed. This is the typical "software engineering" course. Key issues are program efficiency, reliability and maintainability and, typically, these concepts are tied into an introduction to "object oriented analysis," which provides a means of tracing software requirements directly to the implementation.

An "advanced" software engineering course at the graduate level can introduce multi-tasking and management issues, particularly those involving system and program integration.

The above describes a typical approach to the introduction of the concepts of software engineering in a systematic and evolutionary manner. Unfortunately, the key software engineering concepts are not properly conveyed because peripheral issues intervene. The loudest complaint relates to the student having to learn two or three languages in this software engineering cycle. Familiarity with multiple languages is naturally desirable, but not in the software engineering sequence, where consistency and the grasp of the key structured programming concepts are of prime importance.

Typical imperative languages that are used in this sequence are :

- Pascal
- C
- Modula-2

and • Ada.

In addition, functional languages such as LISP and its dialect Scheme have been used, although their usefulness in software engineering is currently still limited.

CURRICULUM DESIGN

When contemplating the development of a software engineering course sequence, it is appropriate to refer to any and all curriculum standards. As an on-going project, the ACM/IEEE-CS Joint Curriculum Task Force [1991] has been predominant in the design of undergraduate curricula and it is important to examine their guidelines when defining course content. One of six curriculum goals is described as follows :

" Undergraduate programs should prepare students to apply their knowledge to specific, constrained problems and produce solutions. This includes the ability to define a problem clearly; to determine its tractability; to determine when consultation with outside experts is appropriate; to evaluate and choose an appropriate solution strategy; to study, specify, design, implement, test, modify and document that solution to evaluate alternatives and perform risk analysis on that design; to integrate alternative technologies into that solution; and to communicate that solution to colleagues, professionals in other fields and the general public. This also includes the ability to work within a team environment throughout the entire problem-solving process"

The subject matter to satisfy the above goal is specified in two subject areas. These are Algorithms and Data Structures and Software Methodology and Engineering. The first area deals with specific classes of problems and their efficient solutions whilst the second deals with the specification, design and production of large software systems.

The subject matter is designed to be presented as a set of "knowledge units" that are the common requirements for a given subject area. Here are, in summary, the knowledge units for the two relevant subject areas.

ALGORITHMS AND DATA STRUCTURES

Basic Data Structures A1
Introduction to lists, arrays, tables, stacks, queues, trees and graphs and their implementations.

Abstract Data Types A2
The purpose and implementation of abstract data types in higher-order languages in the context of large problems, conceptual and formal models , levels of abstraction and reuse of code.

Recursive Algorithms A3
Introduction to the foundations and uses of recursive algorithms in problem solving.

Complexity Analysis A4
Introduction to the notion of computational complexity (time and space) and its use in the analysis of algorithms.

Complexity Classes A5
A general overview of complexity classes P and NP including bound analysis.

Sorting and Searching A6
Comparison of various sort algorithms with a focus on complexity issues and space versus time trade-offs.

Computability and Undecidability A7
Models of computable and undecidable problems including total and partial recursive functions, Church's thesis and universal machines.

Problem-Solving Strategies A8
Introduction to strategies for constructing algorithmic problem solutions for graph and matrix operations,etc. Design and implementations of greedy, divide-and-conquer and backtracking algorithms.

Parallel and Distributed Algorithms A9
Development of algorithms for parallel and distributed architectures.

SOFTWARE METHODOLOGY AND ENGINEERING

Fundamental Problem Solving Concepts S1
Basic ideas of algorithmic problem solving using top-down design, stepwise refinement and procedural abstraction. Basic control structures, data types and I/O conventions.

The Software Development Process S2
Use of tools and environments that facilitate the design and implementation of large software systems taking into consideration the software life-cycle, design objectives, documentation, reliability and maintenance.

Software Requirements and Specifications S3
Development of formal and informal specifications for defining software system requirements.

Software Design and Implementation S4
The design and implementation of large software systems using functional design, bottom-up and top-down design, and consideration for implementation strategies.

Verification and Validation S5
Techniques to verify and validate software systems.

DESIGN OF A COURSE SEQUENCE

Once the above "knowledge units" have been defined in detail , they need to be combined into a suitable suite of courses. The mapping of these units to courses is naturally a subjective process and the particular mapping used happens to be the one adopted at my institution. To assist in understanding the process each knowledge unit has been given a code that defines its content. The Algorithm units are A1 to A9 while the Software units are S1 to S5.

The groupings and course headings appear as follows :

CS101 Freshman Level
Problem Solving & Programming I
A1 S1 S2 S3

CS201 Sophomore Level
Problem Solving & Programming II
A1 A2 A3 A6 A8
S4

CS352 Junior Level
Data Structures & Algorithm Analysis
A1 A2 A4 A5 A6
A8

CS451 Senior Level
Software Engineering
S1 S2 S3 S4 S5

CS493 Senior Level Elective
Computability, Formal Languages & Automata
A2 A3 A4 A5 A7
A8 A9

It should be noted that not all "knowledge units" are of the same length, in terms of course content. Also, many units are spread over a number of courses. The presence of a unit under a course heading indicates that the student is at least provided with an awareness of some of the subject matter. However, the unit number appearing in bold typeface indicates where the greatest emphasis is placed on that unit.

In our program, the first four courses are required whilst the Computability, Formal Languages and Automata course is a senior elective.

Apart from the undergraduate program, there is an advanced software engineering course at the graduate level. This course is specifically designed to incorporate advanced programming techniques and more in depth software engineering concepts. In terms of programming concepts, the course focuses on the following topics

- generic program units
- tasking and concurrency
- exception handling
- packages
- object-oriented design

THE LANGUAGE PHILOSOPHY

Given the above course sequence, it is relevant to examine the desirable features that a language should possess in order to be an appropriate vehicle for the software engineering sequence.

The primary goal of a programming language is to assist the programmer in the software development process. There are a number of characteristics that contribute to this goal, however, fundamentally, a program may be viewed as either a "model of a process" or " a set of actions that we want the computer to carry out." Which view we take, will have an impact on the importance of the characteristics.

The characteristics may be described as follows:

Abstraction

Abstraction relates being able to handle computational objects at varying levels that reveal a certain level of detail. This allows for the suppression of irrelevant details so that the programmer may build modular code that is implementable, modifiable and reusable.

Expressiveness

This refers to how closely the language may represent real world objects and procedures. Expressiveness also relates to how well the language matches itself to a problem domain.

Orthogonality

This relates to the interaction between programming concepts. A truly orthogonal language reduces the inconsistencies that might exist between interacting concepts and allows for a unique means of expressing a given concept in a variety of computational contexts.

Portability

This relates to the ease by which a program may be moved from one virtual machine to another. The portability of a language is greater when a machine-independent standard exists for that language.

Simplicity

The language should be simple from the standpoint of semantics and syntax. Semantic simplicity implies a minimum set of concepts and structures. Syntactic simplicity means that the syntax represents each concept in a unique manner that balances conciseness and readability.

In addition to the above, there are other more pragmatic criteria that will influence the choice of a language. Two additional features relate to the anticipated programmer knowledge Kushan [1993] and the programming environment that supports the language. In selecting a language for an introductory software engineering class, it would be relevant to understand what the anticipated "knowledge" of the students would be.

In relation to the programming environment, features such as context-sensitive editors, symbolic debuggers, program libraries, graphical user interfaces and any other software development tools would enhance the usability of the language as a software development tool.

COURSE DESIGN

The criteria for a suitable software engineering language have been established (Kushan [1993]) and in summary the language should impart the following features:

- syntactic knowledge
- semantic knowledge
- structured programming
- design tools
- debugging techniques

These features should be provided using the characteristics of abstraction, expressiveness, orthogonality, portability and simplicity.

The leitmotif or underlying theme for the course sequence should be the gradual introduction of structured programming concepts and an ability to translate real-world problems into a sequence of programmable actions.

Through the course sequence, the relative emphasis of these two themes will change. In the introductory courses, the primary emphasis will be on the specification of formal problems using either analysis or synthesis. In a problem involving analysis, the initial and final conditions are known but not the specific plan of action. In a problem involving synthesis, the initial condition is known and a specific plan but not the result.

The solutions to formal problems reside in the development of algorithms. In general, the problem must be first understood, then a plan for its solution must be devised, carried out and checked. A key to the successful development of algorithms is the recognition of "algorithmic structures" that are found within the problem definition. These are translated into logical constructs which are defined within the syntax of the programming language.

Next, modules are developed which are collections of logical constructs which have been assembled into a coherent computational process. This introduces the concept of abstraction which allows for the reuse of modules without knowing the underlying details of the logical constructs.

At this point, composite data types are generally introduced which entail the incorporation of arrays and records into problem solutions. By now, all the programming tools should be in place and the emphasis may shift to building structured programs with efficient algorithms.

The sequence of courses now terminates with a traditional software engineering course including design tools and debugging techniques.

ADA: THE LEITMOTIF

Choosing Ada as a unique language for a five course sequence in software engineering is natural. What is most critical however, is the way in which the features of Ada are revealed to the student. A language that is so rich in features can turn out to be confusing and mind-boggling at an introductory level. However, the language may be taught initially in an abstract form, relating to the suppression of irrelevant details.

Freshman Level Course

Typically, students entering an introductory problem solving and programming class will have had exposure to one of two languages, if at all. These are Basic or Pascal. Either of these languages are useful as a prerequisite, particularly Pascal as this may be classed as a sub-set of Ada.

In the introductory course, algorithms may be expressed as "black-box objects" or "sausage machines." In computer science parlance, these are abstract-state machines or automata. The student may act upon the box, by applying a procedure, or examine the contents of the box, by using a function. The student may be taught how to recognize procedures and functions and how to express them in an abstract form. An abstract-state machine becomes a collection of program units.

A typical example is found in Booch [1987] , which addresses the concept at a higher level.

```
package    FURNACE      is
    procedure  SET(TEMPERATURE : in
            FLOAT);
    procedure  SHUT_DOWN;
    function   IS_RUNNING return BOOLEAN;
    function    TEMPERATURE_IS return
            FLOAT;
    OVERTEMP : exception
end FURNACE;
```

Here is a true Ada package which can be conceptualized by an introductory computer science student. The concept of an automaton can be introduced as the first approach to problem solving. The student will analyze the problem for procedures and functions which are then collected into an algorithmic package. No attempt to build the procedures and functions is made until the student can adequately analyze a problem using this approach. These procedural and functional abstractions are true building blocks for the student to learn structured programming. The student can in fact compile these specifications even though the package bodies are not yet written. Gradually, the student will be introduced to the Ada syntax so that the lexical constructs can be used to "flesh out" the procedures and functions. Also the student will, at the outset, have the concept of logically related resources being contained within a package. Such packages can then be used and reused by including them in other programs.

Packages introduce the concept of "top-down" design and this concept may then be the starting point for the use of a text such as Feldman [1992] which may truly be used by novice programmers. Ada can thereby adequately support the needs of a freshman level program by addressing the fundamental problem solving concepts and introducing specifications, logical constructs and basic data types.

Sophomore Level Course

The sophomore level course should fundamentally concentrate on the student being able to generate structured solutions to "more difficult" problems and to understand the software life-cycle. Ada readily supports abstract data types, recursion and has a large library of procedures and functions at its disposal. A text such as Feldman [1992] can be continued over into the sophomore course with a "data structures" text such as Feldman [1985], Stubbs [1993] or Weiss [1993] being introduced to cover algorithm analysis and implementation algorithms.

Junior Level Course

At the junior level, the emphasis shifts towards the development of robust, efficient algorithms that can be measured in terms of their performance. This is done by complexity analysis and space versus time trade-offs. Ada is a powerful, industrial strength language that lends itself to a course involving data structures. Ada was designed at a time when data abstraction was becoming popular. The inclusion of packages, exception handling, generics, representation clauses and tasks support this goal. A point of debate is whether the Ada package is the best way to handle abstract data types. Certainly, it provides a feasible way not available in other languages.

Senior Level Course

The senior level course deals with the development of software-intensive systems that solve complex problems. The tools that assist in solving such problems fall under the label "software engineering." An obvious aim in the process of software development is to match the implementation to the requirements. This is complicated by incomplete and changing requirements. Often the requirements change during development.

To reduce the effects of such inevitable changes, the software development environment should support modifiability, efficiency, reliability and understandability of the software. These factors assist in the management of complexity in relation to the software solution. Ada supports all these goals by modularity, internal documentation and robust software techniques. Indeed, Ada can readily support an object-oriented development mode in which problem space objects can be mapped to system modules. This allows for the comparison of object-oriented design with top-down design and data-structure design.

Ada provides a unique environment for software design in that it provides tools for expressing primitive objects and operations. In addition, Ada is extensible so that the user can construct customized abstract objects and operations.(ACM [1982])Ada is sufficiently expressive to implement a solution and to also capture specifications. The existence of packages, tasking and exception handling further this goal.

Senior Level Elective / Graduate

There are sufficient features in Ada to enable it to be used a vehicle for an advanced senior elective or a graduate course. Such a course can include more complex algorithms that deal with graphs and matrix operations. Also automata and parallel algorithms may be included. For the latter, Ada is beautifully suited to model parallel systems by the use of tasking. (Skansholm [1988]) A typical project for such a course would be to model a real-time control system, such as a bank of elevators.

CONCLUSION

Ada is a consistent language, allowing for no sub-sets or super-sets, which recognizes that a language should promote readability, efficiency and modularity. This very consistency makes it a unique contender as a language for software engineering education. The paper has attempted to demonstrate the needs of a course sequence disassociated from any given language. The only serious challenge to a "pascal-like" language is from C/C++ and, despite many powerful features, the C language is inferior in readability.

Ada, although a full-featured language, can indeed be taught in an abstract fashion. A "top-down" approach to the language may be used to selectively introduce increasing levels of complexity to the student. It is hoped that this positive review of Ada will spark its introduction as the language of choice in many computer science curricula, even where the course of study is not oriented to software engineering.

REFERENCES

Association for Computing Machinery (1982) . Reference Manual for the Ada Programming Language. MIL - STD 1815

ACM/IEEE-CS Joint Curriculum Task Force (1991) . Computing Curriculum 1991

Booch,G . (1987). *Software Engineering with Ada.* Benjamin Cummings.

Feldman, M.B. (1985) *Data Structures with Ada.* Reston

Feldman, M.B., Koffman, E.B. (1992) *Ada : Problem Solving and Program Design.* Addison Wesley

Kushan,B. (1993) Teaching Programming and Problem Solving Strategies in High School Coourses Today *Journal of Computer Science Education* 7(4), 28-35.

Kushan, B. (1994). Preparing Programming Teachers *ACM SIGCSE Proceedings 1994* (to be published).

Lockard, J. (1986) Computer programming in the schools. What should be taught ? *Computers in the Schools,* 2(4), 105-113.

Skansholm, J. (1988) *Ada From the Beginning.* Addison Wesley.

Stubbs, D. F., Webre, N.W. (1993) *Data Structures with Abstract Data Types and Ada.* PWS-Kent.

Weiss, M. (1993) *Data Structures and Algorithm Analysis in Ada.* Benjamin Cummings.

ABOUT THE AUTHOR

Dr Ken Blundell, a native of England, received his Ph.D. from the University of Nottingham in 1977. He held a lectureship at the University of the West Indies in Trinidad (1977-79) before joining the University of Missouri in 1979. His formal education is in mechanical engineering and his research interests have concentrated on the application of neural networks and expert systems to decision making in the areas of product design and manufacturing. He has used Ada extensively for its tasking capabilities and has taught software engineering from the freshman to the graduate level. He has over 50 publications and one book.

ADA BEYOND IMPLEMENTATION

Suzanne Smith
Converse College
Spartanburg, South Carolina 29302-0006

Donald Gotterbarn, Robert Riser
East Tennessee State University
Johnson City, Tennessee 37614-0711

ABSTRACT

Ada is more effectively taught if it is seen and used by students as more than just a programming language. This paper describes the use of Ada as an entire life-cycle tool throughout a two-semester undergraduate software engineering course. Ways to demonstrate Ada's relevance in requirements analysis and design are also described. The coordinated blending of lectures and projects and the use of three techniques in teaching Ada - a spiral approach, program reading, and a detailed examination of language features - enables students to see Ada's support for software engineering in and beyond implementation.

I INTRODUCTION

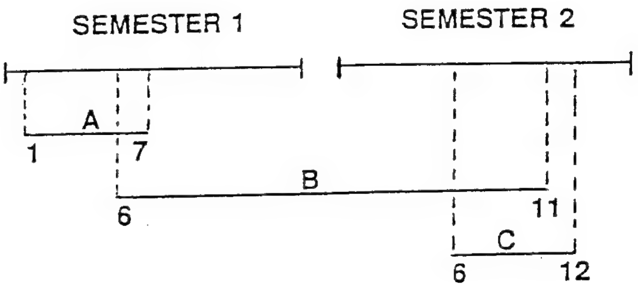
Teaching a programming language typically consists of a detailed examination of language features only in order to proceed with implementation. Ada, however, supports software development beyond implementation. A better approach to teaching Ada is to shift its introduction into earlier phases of the life cycle so as to demonstrate its value as an entire life cycle tool.

Our approach uses three distinct teaching techniques within a two-semester projects-oriented course: a spiral approach, program reading, and a detailed examination of language features. The spiral approach to teaching involves the gradual introduction and application of concepts and the revisiting of them in increasing depth throughout the remainder of the course in both lectures and project activities. For example, when discussing requirements, we briefly mention configuration management techniques. The subject of configuration management is revisited during subsequent phases in increasing detail. Applying this spiral approach, Ada is introduced as software engineering concepts (e.g., abstraction, information hiding, reusability, and maintainability) are introduced. With each concept, the language features of Ada which support that concept are also introduced. For example, during the discussion of maintainability students are shown language features such as named association of parameters, overloading, packages, and separation of interface specifications. These language features are presented initially through program reading utilizing small examples and are discussed only in relation to their support of maintainability. Only after students have experienced Ada as a software development tool do they examine in detail the language features of Ada. Ada's role in the course projects is described in the following section.

II THE PROJECTS-ORIENTED COURSE

We use the spiral approach, program reading, and a detailed examination of language features to teach Ada in an undergraduate two-course software engineering sequence. (Syllabus in Appendix A.) In this undergraduate sequence, we have incorporated experience gained in developing and teaching courses in our graduate software engineering concentration, including the coordinating of some graduate student team milestone reviews in the undergraduate classroom. A DARPA grant enabled us to complete development and implementation of this approach to teaching software engineering which incorporates Ada throughout the life cycle.

A number of different approaches have been taken to incorporate project experience into a two-semester software engineering course. In our approach to teaching software engineering, we incorporated project experience through-out the two semesters. This differs from other approaches. For example, [1] devotes the first semester to theory and the second semester to applying the theory to a project. Others have integrated a large project throughout by concentrating on analysis and design in the first semester followed by implementation and testing in the second. Our approach reflects the iterative nature of software development by providing experience on multiple projects throughout both semesters, as indicated in Figure 1.



A - small project
B - extended project
C - maintenance project

STRUCTURE OF THE PROJECTS
Figure 1

The three projects provide the focus for the two semesters and, collectively, provide students with a variety of experiences and roles. The first project, referred to as the small project, begins in week two and extends through week seven of the first semester. The second project, referred to as the extended project, is introduced in week six of the first semester and extends through week eleven of the second semester. Finally, the third project, referred to as the maintenance project, occupies weeks six through twelve of the second semester. Note that the projects overlap and all are completed prior to the end of the second semester.

In order to provide both varied and realistic experiences, the three projects differ in a number of significant ways: size and complexity, project team organization, information provided to the teams at the start of the project, deliverables produced by the student teams, development methodology, tools, controlling disciplines, and the use of reviews. A brief description of the three projects follows and additional details are provided in [4].

The small project provides an early and quick immersion into the software development process. A problem statement is provided and student teams are expected to rapidly specify, design, code, and test a solution. There is minimal logical complexity in the small project so that student efforts can focus on the details of design and development of the software. Democratic teams of four to six students are used. In addition to the narrative problem statement, a software project management plan (see Appendix B) with scheduled reviews and deliverables is provided, and regular (and documented) team meetings are required. Structured analysis and design techniques are used and implementation is in a language with which the students are already proficient (Pascal). Ada is introduced, not as an implementation language, but as a specification tool used in high level design. Despite the introduction of sound software engineering principles, the project is such that student teams in some cases are able to operate in an *ad hoc* fashion and still achieve a modicum of success.

During the last week of the small project, the students are introduced, via a customer request, to the extended project in which Ada has an expanded role. The project is developed using Ada in the high-level design and as the implementation language. This project serves as a vehicle both to revisit concepts in depth that were briefly introduced in the small project and to introduce and utilize new concepts including detailed design, configuration management, and verification and validation. The extended project requirements are considerably more complex than those of the small project and students are exposed to problems that were not apparent during the small project. Similarly, this project requires the use of more effective controlling disciplines and increased attention to the software process itself. For example, a configuration manager is appointed and a configuration management plan developed and enforced. Another effect of the complexity of the project is the necessity for closer attention to the specification and the design.

The entire class works on the extended project, organized into teams which reflect the division of any software development project into analysis, design, and implementation phases, and support functions. Analysis phase teams include a requirements team, a user interface team, and a test plan team. Design phase teams include a preliminary design team, a detailed design team, and a user interface team. Implementation phase teams include a code team and a testing team, and entire life-cycle support teams include configuration management and tools. In the course of the extended project, all students serve on either multiple teams or on an entire life cycle support team. No student is assigned to two teams which are responsible for validating one another's work (e.g. code team and testing team) and the organization allows teams to act as cross checks on each other during development. For example, the user interface team meets independently with the user, while the requirements team meets with the customer. Thus, during the requirements review, the user interface team can help validate the requirements. Object-oriented design is utilized and the deliverables are discussed in Section III. Again, a project management plan (see Appendix C) is provided, including deliverables (configuration items) and scheduled reviews. The use of reviews is more frequent and extensive, including scheduled milestone reviews and the encouraging of student teams to hold internal reviews (e.g. code reviews).

During the first semester, the extended project is taken through the preliminary design review. Deliverables, specified in Ada, become baseline documents for detailed design beginning in the second semester. Tools which we have developed to facilitate Ada implementation (e.g., modified Nassi-Shneiderman diagrams and object traceability matrix) are introduced during the extended project.

The maintenance project overlaps the last five weeks of the extended project and continues one week beyond it. The overlap is intentional, in order to more accurately reflect a typical industrial environment. Maintenance tasks, originating in the form of a change request approved by a configuration control board, are assigned to student teams. For these maintenance tasks, the students are organized into chief programmer teams. A large Ada artifact is provided and students must implement the change, including modification of all appropriate documents. The maintenance project constitutes yet another circuit in the spiral from which to revisit and reinforce significant software engineering concepts, but this time from the maintenance perspective. Maintenance is treated as a complete software development task, including analysis, design, implementation, and testing. This disciplined approach to maintenance provides an experience rarely achieved in traditional software engineering courses, and coming after their experiences on the small and extended projects, allows students to better understand the benefits of following good software engineering practices.

Thus Ada is integrated into the project experiences as well as into the lectures. Early in the first semester it is introduced through program reading techniques [2], and program reading continues throughout (see Section IV). Other activities include the writing of Ada high-level design specifications, the implementation of a system in Ada, and maintenance on an existing Ada system. This approach to teaching and learning Ada enables students to see Ada as more than an implementation language.

III THE SPIRAL APPROACH

We have developed what we call a "spiral approach" to teaching and learning software engineering in a two-course sequence. We combine a layered approach where "inter-related topics are presented repeatedly in increasing depth" [3] during lectures with a careful coordination of project and lecture stages which emphasizes the relationship of software engineering principles to software development [6]. The combination of these two techniques, coordinating the increase in depth of the lectures with more demanding project experiences, constitutes our spiral approach.

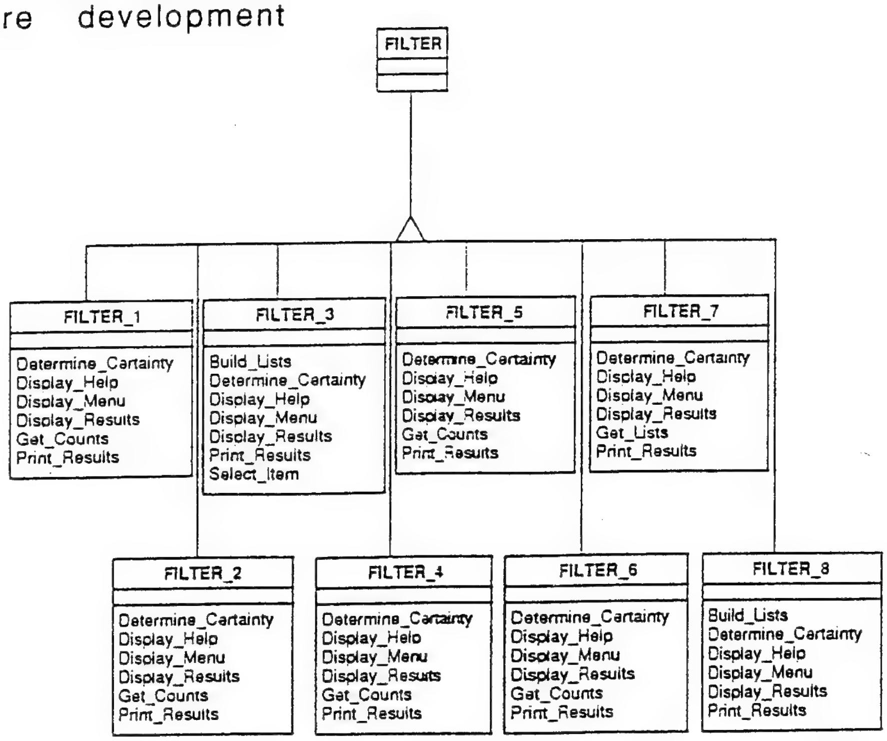
Placing a small project at the beginning of the course permits us to start the spiral approach immediately. Lectures include an rapid overview of the elements of a software development life cycle. We emphasize design and introduce Ada as a specification language through program reading. These concepts are also emphasized in project deliverables through Ada specifications and a preliminary design review. This first pass through the lecture-project spiral imposes controlling disciplines upon the teams with minimum justification during lecture. For example, students are immediately introduced to various life cycle elements, project organization, configuration management, quality assurance, and verification and validation techniques by "living them" but only later are these topics formally addressed in lectures.

An extended project provides an opportunity for another pass through the lecture-project spiral. During this phase of the course all of the concepts from the first pass through the spiral are revisited and expanded upon in both lecture and project deliverables. These concepts and the project processes and deliverables include reviews, controlling techniques, software development standards, Ada as a software development tool, and development team organizations. While Ada was used only in the high-level specification of the small project, it is now used as a requirements specification and design tool as well as an implementation language.

Object-oriented design is used in the extended project. The required preliminary design deliverables include:

1. An object diagram using Rumbaugh notation [5],
2. A class dictionary,
3. An object-requirements traceability matrix,
4. Ada specifications for each object class, and
5. Descriptions of all major user interfaces.

The object diagram (Figure 2) is produced using OMTool. (A trademark of GE Advanced Concepts.)



OBJECT DIAGRAM
Figure 2

We have designed a class dictionary (Figure 3) to supplement Rumbaugh's object model.

The class dictionary entries include a description of each object, its attributes and methods, and its interfaces with other objects. The use of this form forces students to address the interfaces between objects and define their associations early in the design process. To establish

OBJECT CLASS NAME: Filter_3

OBJECT DESCRIPTION:

- Examines the function and procedure interfaces of Pascal source programs and generates the lists for Filters 4 and 5.

ATTRIBUTE DESCRIPTION:

- **Certainty** - Measures how similar Pascal source programs are, by comparing the function and procedure interfaces.
- **ListoNames_Parms_1** - A list of function / procedure names and counts of their parameters by type in Pascal source program 1.
- **ListoNames_Parms_2** - A list of function / procedure names and counts of their parameters by type in Pascal source program 2.

METHOD DESCRIPTION:

- **Build_List** - Build a list of all function and procedure names, their parameter types and an indication of which ones are VAR parameters.
- **Determine_Certainty** - See Process Specification 2.3.3
- **Display_Help** - Displays the help information for this Filter.
- **Display_Menu** - Displays the menu information for this Filter and returns the action the user wishes to perform.
- **Display_Results** - Displays the results of this Filter.
- **Print_Results** - Prints the results of this Filter to the summary report.
- **Select_Item** - Displays all the items in a list and returns the one the user selected.

INPUT INFORMATION NEEDED FROM OTHER OBJECTS:

- **MENU.DISLAY_FILTER_3_MENU => Integer** - The return code from this routine is an integer which is the action the user wishes to be performed (either run, help or exit).
- **SOURCE_PROGRAM.Gen_Function_Parms => Integer** - Generates a list of functions and their parameter list.
- **SOURCE_PROGRAM.Gen_Procedure_Parms => Integer** - Generates a list of functions and their parameter list.

CLASS DICTIONARY
Figure 3

traceability between software development phases, we have also designed an object traceability matrix (Figure 4). This matrix provides a backward trace from each design object to a specific requirement (preliminary design traced to requirements) and a forward trace from each design object to a specific Ada specification (preliminary design traced to detailed design). The introduction of this form allows us to revisit the concept of traceability in the software life cycle.

Functional Requirement	Object Name	Ada Package
1,2,10,11	FILE	File
20,21,22	SOURCE_PROGRAM	Source_Program
3,4	MENU	Menu
5,6	HELP	Help
7,8,9,19,23,24, 25,26,27	FILTER_1	Filter_1
7,8,9,19,28,29, 30,31,32,33,34	FILTER_2	Filter_2
7,8,9,19,35,36, 37,38	FILTER_3	Filter_3
7,8,9,19,39,40, 41,42,43,44,45, 46	FILTER_4	Filter_4
7,8,9,19,39,40, 41,47,48,49,50, 51,52,53,54,55	FILTER_5	Filter_5
7,8,9,19,55,56	FILTER_6	Filter_6
7,8,9,19,57,58, 59,60	FILTER_7	Filter_7
7,8,9,19,61,62, 63	FILTER_8	Filter_8
12,13,14,15,16, 17,18	SUMMARY	Summary_Report

OBJECT TRACEABILITY MATRIX
Figure 4

Traceability is extended into detailed design by means of a detailed design traceability matrix (Figure 5). We created this matrix to provide traceability between preliminary design, detailed design and implementation. The detailed design matrix first provides traceability between an object's attributes and its data structures and between those data structures and their Ada package representation. The matrix also provides traceability between the object's operations, the detailed design model of those operations and the Ada package embodying those operations.

We used Nassi-Shneiderman diagrams as the detailed design model for these operations. Nassi-Shneiderman diagrams provide a code independent notation for the development of algorithms which allows the students to think through the solution without considering language details. We created Nassi-Shneiderman extensions to include language features of Ada which are unique and which have no associated notation. Figure 6 shows the extension of the notation for an exception handler.

During the extended project, program reading is continued as examples and classroom exercises provided go into greater depth. Carefully organized use of self-paced Ada tutorial materials, laboratory experiences, and the Ada Language

DATA STRUCTURES

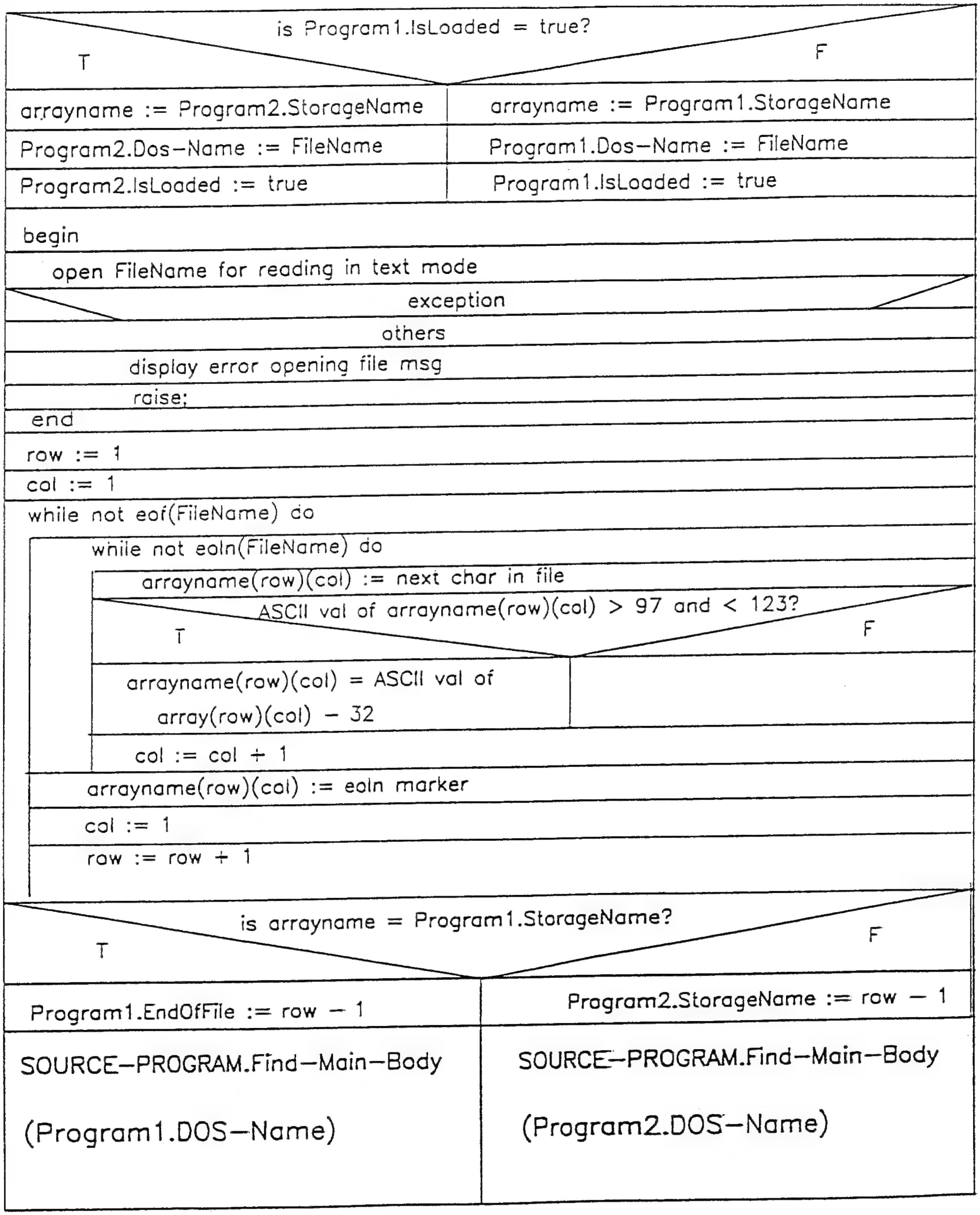
Data Structure Name	*Objects* & Attributes	Ada Package
FILTER3_DATA	*FILTER_3* ListONames_Parms_1, ListONames_Parms_2	PD_FILT3.ADS
FILTER3_RESULTS	*FILTER_3* FILTER3_DATA Certainty	PD_FILT3.ADS
F4_Comparisons	*FILTER_3*	PD_FILT3.ADS
F5_Comparisons	*FILTER_3*	PD_FILT3.ADS

OPERATIONS

Nassi-Shneiderman Model Name	Object_Operations	Ada Package
Build-List	Build_List	Filter_3
Determine-Certainty	Determine_Certainty	Filter_3
Display-Menu	Display_Menu	Filter_3
Display-Help	Display_Help	Filter_3
Display-Results	Display_Results	Filter_3
F4-Has-Been-Compared	not specified	Filter_3
F5-Has-Been-Compared	not specified	Filter_3
Initialize-F4-Comparisons	not specified	Filter_3
Initialize-F5-Comparisons	not specified	Filter_3
Max-Comparisons	not specified	Filter_3
Print-Results	Print_Results	Filter_3
Select-Item	Select_Item	Filter_3

DETAILED DESIGN TRACEABILITY MATRIX
Figure 5

procedure Source-Program.Load-File-Into-Memory(FileName : in string);



```
is Program1.IsLoaded = true?
  T:
    arrayname := Program2.StorageName
    Program2.Dos-Name := FileName
    Program2.IsLoaded := true
  F:
    arrayname := Program1.StorageName
    Program1.Dos-Name := FileName
    Program1.IsLoaded := true
begin
  open FileName for reading in text mode
exception
  others
      display error opening file msg
      raise;
end
row := 1
col := 1
while not eof(FileName) do
    while not eoln(FileName) do
        arrayname(row)(col) := next char in file
        ASCII val of arrayname(row)(col) > 97 and < 123?
          T:
            arrayname(row)(col) = ASCII val of
              array(row)(col) - 32
          F:
        col := col + 1
    arrayname(row)(col) := eoln marker
    col := 1
    row := row + 1
is arrayname = Program1.StorageName?
  T:
    Program1.EndOfFile := row - 1
    SOURCE-PROGRAM.Find-Main-Body
    (Program1.DOS-Name)
  F:
    Program2.StorageName := row - 1
    SOURCE-PROGRAM.Find-Main-Body
    (Program2.DOS-Name)
```

NASSI-SHNEIDERMAN CHART
Figure 6

Reference Manual makes it possible to minimize in-class discussion of Ada syntax. In lectures, Ada's role in specification and design is emphasized. The use of accepted controlling techniques and standards introduced in the small project is also reinforced.

The maintenance project helps students see the utility of controlling techniques during original development. Ada's impact on the development of maintainable artifacts is emphasized. By equating maintenance and development, the students revisit most of the concepts previously discussed. This third trip through the spiral makes it easier for them to work with a large unfamiliar artifact. Many students find this somewhat surprising and rewarding.

We have found this lecture-project spiral approach to be an effective teaching and learning technique. The project framework provides a series of passes through the software development process, each pass adding to a body of common experiences to which subsequent passes can refer. By the middle of the first semester students, individually and in teams have their own "war stories". This personalized knowledge provides a solid base for more advanced concepts [4].

IV PROGRAM READING

Program reading is an effective technique in teaching Ada because it allows the students to see software systems developed in Ada before they worry about the implementation details of the language. In program reading, students become familiar with a programming language through first reading programs rather than writing programs. Relatively large, well-designed software artifacts written in Ada are examined by the students. The students see not only the source code but also supporting documentation (e.g., analysis model, design model). Program reading emphasizes the design of the system so that the students get the "big picture" of how effective it is to build software using Ada. Language features are pointed out but discussed only in relation to effective software design. For example, as the software engineering concept of abstraction is discussed, its support in Ada is examined through program reading. Program reading is also consistent with the spiral approach since it allows the software to be examined in increasing detail as the students' knowledge increases.

V DETAILED EXAMINATION OF LANGUAGE

The detailed examination of language features is the last technique used to teach Ada. Only after students have experienced Ada as an analysis and design tool are they introduced to Ada as an implementation tool. This approach is particularly applicable to Ada because of its support throughout the life cycle while most other programming languages are merely implementation tools. Most of the language features examined at this time have already been demonstrated to the students through the program reading or examples which supported the software engineering concepts and topics discussed in the lectures. Because of this familiarity with the language, the students move very quickly through the implementation details.

VI CONCLUSION

The teaching techniques used - the spiral approach, program reading and detail examination of language features - provide an effective way to teach Ada. This approach and the tools developed for this project have effectively been used to demonstrate the relationship of Ada to software engineering concepts and Ada as an entire life cycle support tool. Students learn to fully appreciate and effectively utilize Ada only when they see Ada beyond implementation.

ACKNOWLEDGEMENTS

This Project was partially funded by DARPA research grant DAAL-03-92-G-0411. We would like to thank the Defense Advanced Research Project Office and the Ada Joint Program Office for their support of the improvement of software engineering curricula and the support by the U.S. Army Research Office.

REFERENCES

1. E. Adams, "Experiences in Teaching a Project-Intensive Software Design Course," Proceedings of the First Annual Rocky Mountain Small College Computing Conference, volume 8, number 4, March 1993, pp. 112-121.

2. L.E. Deimel and J.F. Neveda, "Reading Computer Programs: Instructor's Guide and Exercises," CMU/SEI-90-EM-3.

3. G. Ford, N. Gibbs, and J. Tomayko, "Software Engineering Education: An Interim Report from the Software Engineering Institute," SEI-87-TR-8.

4. D. Gotterbarn, R. Riser, "Real-world Software Engineering: A spiral approach to a project oriented software engineering course," in Software Engineering Education, ed. Jorge L. Diaz-Herrera (Springer-Verlag, New, N.Y. 1994)

5. J. Rumbaugh, et. al., Object-Oriented Modeling and Design, (Prentice Hall, NJ) 1991.

6. M. Shaw and J. Tomayko, "Models for Undergraduate Project Courses in Software Engineering," Software Engineering Education, SEI Conference 1991, Pittsburgh, PA, October 7-8, 1991, Springer-Verlag, New York, NY).

BIOGRAPHIES

Suzanne Smith is an assistant professor of computer science at Converse College and has previously taught at East Tennessee State University. She has also worked as a software engineer at Lockheed Missiles & Space and as a visiting professor at the Software Engineering Institute. She holds a PhD from Florida State University.

Robert Riser is an associate professor of computer and information sciences at East Tennessee State University. He has previously worked as a member of the technical staff at Bell Telephone Laboratories. He holds an MS from Stevens Institute of Technology.

Donald Gotterbarn is an associate professor of computer and information sciences and coordinator of the graduate software engineering concentration at East Tennessee State University. He has also worked as a computer consultant and as a visiting scientist at the Software Engineering Institute. He holds a PhD from the University of Rochester.

APPENDIX A

SYLLABUS - FIRST SEMESTER

Week 1: Introduction to software engineering
- a) Definition, scope, objectives; relationship between computer science and software engineering; software as an engineered product.
- b) Current state of software engineering

Overview of the life cycle and controlling disciplines
- a) Life cycle phases; objectives, activities, deliverables
- b) Life cycle models
- c) Qualities of well engineered software

Introduction to team projects
- a) Project leadership; working in groups
- b) Team organization, meetings, techniques
- c) Team formation; initial meetings, establishment of meeting times

Written and oral communication skills

Week 2: Overview of Controlling Disciplines
- a) Configuration management
- b) Software quality assurance
- c) Software verification and validation
- d) Relationship of a-c to team project 1

Presentation of Requirements for first team project
- a) Assignment of projects to teams
- b) Project management plan; project deliverables
- c) Test plans, verification and validation for projects

Week 3: Introduction to software design: purpose, process, reviews
Structured design: methods, notations, tools
Introduction to ADA; ADA as a design notation
Team presentations of small project designs
Design standards: structural and environmental

Week 4: Object-oriented design
- a) Introduction to object-orientation
- b) Review project as object-oriented
- c) Examples of project one objects in Ada and their relation to structured design

Week 5: Verification for real projects
- a) Software quality assurance
- b) Software testing: overview, types, strategies
- c) Software safety

Week 6: Preliminary Description of Acceptance Reviews
- a) Format and content of reviews
- b) Professional responsibility for product

Requirements engineering
- a) Elicitation of requirements; identifying and interacting with users
- b) Requirements analysis

Introduction to project 2: the client request

Week 7: Team presentations and reviews of project one; submit specifications, designs, test histories, instruction manual, documented code and executable system

APPENDIX A - Page 2

First Major Examination
Announce requirements team assignment and set up preliminary customer interview

Week 8: Review Examination

Project 2 organization
a) Matrix model: team roles,responsibilities, and deliverables
b) Formation of teams; initial team meetings and establishment of meeting times

Continue discussion of requirements engineering
a) Requirements specification: methods and notations
b) Ada as a specification language

Week 9: Continue discussion of requirements engineering
c) Logical vs physical views
d) CASE tools for analysis
e) Analysis documents
f) Assessment activities and role of controlling disciplines (CM, SQA, V&V) in requirements engineering
Presentation of Configuration Management Plan

Week 10: Requirements reviews of project two
a) Requirements team presentation
b) First drafts of user manual and of test plan, baselined requirements placed under configuration control
Introduce software development planning and milestones
Introduce systems design
a) Design methods and strategies

Week 11: Continue discussion of systems design
b) Characteristics of good design
c) Human engineering standards
d) Transition from analysis
e) Design fundamentals: modularity, structure, abstraction, information hiding

Week 12: Continue discussion of systems design
f) Design methods and tools: structured design, object-oriented design, other design methodologies
g) Ada and systems design
Second Major Examination

Week 13: Review of Examination
Continue discussion of systems design
h) Preliminary design
i) Detailed design
j) Design documents
k) Standards- 2167a
l) Assessment activities and role of controlling disciplines (CM, SQA, V&V) in design

Week 14: Design Review of project two
a) Design team presentation
b) DUE: 2nd draft of user manual; 2nd draft of test plan
Submission of Baselined Design for Project two
User manual, test plan and design placed under configuration control

Week 15: Final Preliminary design due
Faculty assessment of final preliminary designs
Professional and moral issues in design decisions

APPENDIX A - Page 3

SYLLABUS - SECOND SEMESTER

Week 1: Introduction and review of semester 1
Review Project 2 specification and preliminary design documents (Requirements analysis and specification through preliminary design completed in 1st semester)
Reorganize project 2
a) Matrix model: team roles, responsibilities, and deliverables
b) Formation of teams: initial meetings, establishment of meeting times

Week 2: Detailed Design
a) Methods
b) Notations
c) Tools
d) Goals: flexibility, maintainability, cost, reliability, testability, reusability, etc.
Assessment: Formal and Informal Reviews {V&V}
a) Software quality assurance
b) Walkthroughs
c) Inspections

Week 3: Software implementation
a) Programming practices; standards
b) Programming environments and tools
c) Ada specifications and implementation
d) Comparison of implementation languages
Student walkthrough of detailed design

Week 4: Detailed Design Review of project two
a) Design team presentation
b) Submission of draft version 3 of user manual
c) Submission of draft version 3 of test plan
Submission of baselined detailed design for project two
Revised user manual, test plan and design placed under configuration control
Software Implementation, continued
d) Comparison of implementation languages
e) Assessment activities and role of controlling disciplines (CM, SQA, V&V) in implementation

Week 5: Final Detailed design due
Acceptance test plan due
Software project management
a) Software metrics
b) Estimation and scheduling
c) Organizational structures
d) Project management plans
e) Professional and moral issues in project planning
Faculty assessment of final detailed design

Week 6: Software evolution/maintenance
a) Evolution/maintenance considerations throughout the life cycle
b) Types of maintenance: corrective, perfective, adaptive
c) Maintenance tools and techniques
d) role of controlling disciplines, change requests, discrepancy reports
Maintenance project and team assignments

Week 7: First major examination
Software evolution/maintenance, continued
e) Assessment activities and role of controlling disciplines (CM, SQA, V&V) in maintenance

Week 8: Code inspections
Software testing and maintenance
a) regression testing
b) integration testing
Team code inspections and reports due

Week 9: Unit testing and advanced test case design
Testing tools and test generators
User interface development
a) design and human factors
b) tools

Week 10: System test
Acceptance test presentations
Real-time systems
a) design
b) implementation and testing considerations

Week 11: Final Project 2 deliverables
Maintenance Project Due
Revisit Assessment activities and role of controlling disciplines (CM, SQA, V&V) in maintenance
Software safety: design, languages, maintenance, useability related to project 2

Week 12: Second Major Examination
Review Examination
Beginning of assessment period for all projects
Software Product Metrics using Project 2
a) Complexity
b) Size
c) Estimation and scheduling

Week 13: Software Product Metrics
d) Testing
e) Reusability
f) CASE tools
Results of using metrics tools on project 2

Week 14: Software Project Metrics
a) Estimation and scheduling
b) Staffing

Week 15: Legal and ethical issues, professionalism
a) Professional ethics and responsibilities
b) Ethical issues
c) Professionalism
Lessons learned document due

APPENDIX B

Software Project Management Plan - Small Project

Week, Class	CI Id	Description
1a		Requirements statement distributed
1b	CI-1	Requirements: abstract of project and detailed list of requirements
2b	CI-2	Analysis decisions completed: CD, DFD, and data dictionary
3a	CI-3	Design documents: system architecture - structure chart and external descriptions of modules and interfaces
	CI-4	Test plan: classes of tests for each requirement
3b		Presentation of design review
		Modify design and check coding standards
4a		Begin coding system and design of test cases
		Detailed architectural reviews
5a	CI-5	Test cases: specific tests, their input and expected output and their relation to requirements
		Code reviews and unit tests and corrections
6a		System testing and corrections to program
6b	CI-6	Documented source code
	CI-7	Executable code
7a	CI-8	Certified Acceptance Test: documentation of test cases and their relation to test plan; documentation of consistency of source code structure with architectural design; also include package of CI-1 through CI-6
		Presentation of system to customer

NOTE: **All presentation/review items are distributed to designated reviewers 24 hours prior to the presentation/review.**

APPENDIX C

Software Project Management Plan
Extended Project, Semester 1 (through Preliminary Design)

Week, Class	CI Id	Description
1a		Customer request presented.
1b		Team assignments announced and roles defined.
		Start development of configuration management plan (CMP), preliminary requirements (P_REQ), preliminary test plan (P_TP), and preliminary user's manual (P_UM).
3a	CI-1	CMP delivered and presentation to teams.
4a	CI-2	P_REQ delivered and presentation to teams and customer.
4b	CI-3	P_TP delivered and presentation to teams.
	CI-4	P_UM delivered and presentation to teams.
5a		Requirements review. Preliminary design begins.
5b	CI-5	Final revised requirements delivered and baselined.
7a		Preliminary design review. Detailed design begins.
9a	CI-6	Final preliminary design delivered and baselined.
9b	CI-7	Final test plan delivered and baselined.
9b	CI-8	Final user manual delivered and baselined.
10		Milestone acceptance review.

NOTE: **All presentation/review items are distributed to designated reviewers 24 hours prior to the presentation/review.**

AN ADA/SOFTWARE ENGINEERING MIGRATION TRAINING PLAN

James W. Hooper, Akhtar Lodgher, Hisham Al-Haddad

Marshall University
Department of Computer Science and Software Development
Huntington, West Virginia 25755
(304) 696-5424

Keywords: Ada Training, Software Engineering Education, and Software Engineering Curriculum

Summary

In many companies and government agencies, previous experience has failed to adequately prepare personnel to deal with the complexities of their current software development and maintenance needs. A number of organizations have undertaken extensive reorientation and retraining efforts to prepare for more effective software engineering practice. In the work reported here, the authors surveyed government, industry, and academia regarding past experiences in retraining personnel. Identification has been made of required knowledge and skills to effectively develop and maintain software, emphasizing the Ada language. Modules of software engineering knowledge and skills are outlined based on identified requirements. Suggestions for curriculum content and implementation are provided.

1. Introduction

This paper presents a summary of work performed for the U.S. Army Corps of Engineers[1]. The procedure followed was to survey previous experience in retraining personnel, identify successful approaches, identify knowledge and skills needed for success, and formulate a curriculum to help develop the needed knowledge and skills.

The paper is organized as follows: Section 2 documents the survey of government, industry, and academia regarding past experiences in retraining personnel. Section 3 provides an identification of required knowledge and skills to effectively develop and maintain software using the Ada language. Section 4 highlights proposed curriculum content and offers suggestions for implementing the curriculum, making use of insights gained in the survey of experiences. Section 5 is the conclusion.

2. Survey of experiences in Software Engineering/Ada Training and Education

Many training/retraining experiences have been documented and/or presented in conferences and other public forums. Other experiences were known to participants in this study, or to other professional acquaintances. We have selected a few experiences in each of the categories of government, industry, and academia, which collectively represent a rather broad range of personnel backgrounds and training/education approaches, and include aspects worth considering relative to the training needs of many organizations. In the following subsections we present an overview of the training and education experiences selected.

[1] Under Contract No. DACW69-90-0012, Mitchell Systems Corporation, Work Order No. MSC3-024.

2.1 Government

Naval Command, Control, and Ocean Surveillance Center: The Research and Development, Test, and Evaluation Division (NRaD) of the Naval Command, Control, and Ocean Surveillance Center (NCCOSC) (formerly the Naval Ocean Systems Center), San Diego, CA, undertook to improve their software process following a Software Engineering Institute (SEI) process assessment in February 1988. NRaD was found to be a Level 1 organization. Nrad's work force includes about 3,500 civilians (of which 1,800 are scientists and engineers) and 350 military personnel. The Software Engineering Process Office (SEPO) was formed in November 1988, with the mission to improve overall quality of software engineering processes and software products at NRaD. A paper by Duston Hayward [1] describes NRaD's efforts. We also communicated directly with Mr. Hayward.

SEPO is responsible for defining and improving software processes, so it was considered to be the appropriate group to plan and offer training (based on the process), provide consultation on software implementation projects, and measure process improvement. Members of the SEPO (Hayward and others) planned and are teaching the NRaD Software Project Management (SPM) course. The SPM course targets two key areas of learning needed: cost and schedule estimation models, and understanding of software development. The course as planned spends 58% of the time in requirements and planning of software project; 12% in software estimation, 11% in formal inspections, and 10% in project metrics; 60% of the time is spent in instruction, and 40% in exercises.

The SPM course objectives are: Develop and improve skills required to plan and manage software projects, understand NRaD's role in software development, deepen understanding of DOD-STD-2167A throughout the life cycle, provide solid understanding of early phases of the life cycle, improve software development process and maturity at NRaD, and give real-life examples through case studies. Although about 90% of software is developed by contractors and 10% in-house, they emphasize the need for all management to fully understand the concepts of process and project management.

SEPO spent 2,043 hours developing the course and fine tuning it prior to the first offering in January 1992. They are approved for 600 hours per year of effort for continuous improvements in the training course. In follow-up evaluations it was found that significant percentages of managers adopted concepts learned in the course. In particular, they have found that the use of formal inspections has reduced the time to find and fix defects, reducing cost from thousands of dollars per defect to about $100 per defect.

NRaD is taking the lead in the Navy for software process improvement. They have also taught Air Force, Army, and SEI personnel, and are planning a new eight-hour course targeted to senior managers. They plan to offer a train-the-trainer SPM course also. In addition to their SPM course, courses are offered in object-oriented analysis and object-oriented design by a contractor (Project Technology), based on the Schlaer-Mellor method. Ada courses are taught by Air Force officers from Keesler Air Force Base (80 hours instruction).

USACE TEXEL Training: The U.S. Army Corps of Engineers (USACE) Waterways Experiment Station (WES) in Vicksburg, Mississippi, planned for vendor-taught training in software engineering and Ada. Memphis State University, under sponsorship of WES, recommended the following curriculum:

1. A two-day overview for technical management and staff.
2. A five-day programmer's introduction to the Ada language
3. A five-day advanced sequel to 2.
4. A five-day course on Ada software engineering concepts
5. A five-day course on object-oriented design technique for Ada

TEXEL & COMPANY offered four weeks of training. Dr. O.E. Wheeler of Memphis State taught some, but not all, of the attendees a two-day overview course prior to the TEXEL training. Training actually offered by TEXEL was four one-week classes, in the following sequence: Object-Oriented Design with Ada, Introduction to Ada, Advanced Programming with Ada, and Object-oriented Analysis.

Dr. Bill Ward (of the University of South Alabama), who also worked with WES in this training, noted that some, but not all, of the attendees knew a high-level language. Some knew something about Ada, while some had only COBOL experience. Some of the participants were recent college graduates. The support tools used in the training were an Alsys Ada compiler, and "Ada World"--a Turbo-Pascal-like environment. Dr. Ward feels that a greater use of CASE tools is needed.

U.S. Army Materiel Command Intern Program: The Army Materiel Command (AMC) Software Engineering Intern Program was established in 1986 to provide a constant source of software engineers for AMC [2]. The program has concentrations in software engineering concepts, the Ada programming language, mission critical computer systems (MCCS) software, and Army systems acquisitions concepts. The program consists of two years of study, the first year at the AMC School of Engineering and Logistics at Red River Army Depot, Texarkana, Texas, and the second year at Monmouth College, West Long Branch, New Jersey, and the U.S. Army Communications Electronics Command (CECOM), Ft. Monmouth, New Jersey. Currently all students in this program must have an undergraduate degree in engineering.

The year at the AMC School in Texas consists of five terms of 9-10 weeks each. Groups of courses are offered in a variety of areas including Software Engineering Concepts, The Ada Programming Language, MCCS Software Requirements, Army Systems Acquisition Requirements, and Acquisition Management. During the second year of the program, 50% of the time is spent in on-the-job training at CECOM's Software Engineering Directorate, and 50% in formal academic courses in Monmouth College's M.S. in Software Engineering program.

Arrangements are currently under consideration to create a similar arrangement by teaming the U.S. Army Missile Command in Huntsville, Alabama with an M.S. program in software engineering at the University of Alabama in Huntsville. Consideration is being given to undertake retraining current government engineers, and to modify the program for non-engineer candidates. Also additional research and consulting by the faculty is a consideration.

2.2 Industry

Grumman Melbourne Systems Division: Grumman Melbourne Systems Division (GMS) in Melbourne, Florida, undertook their current training effort in 1991. According to the GMS document GMSD-SW-PLAN-002, "GMS is a software intensive organization, and has adopted an aggressive training agenda that will support the transition to a well-disciplined software engineering technology at GMS." GMS undertook to develop a training plan in view of the SEI Capability Maturity Model (CMM), and the training needs of each CMM level. They developed a seven track training program, with tracks as follows:

1. Software Project Managers: i.e., software engineering managers
2. Technical Management: line management (project leaders, group leaders)
3. Support Management: Software Quality Assurance (SQA), Software Configuration Management (SCM), Integrated Logistics Support (ILS), and Test Managers
4. Technical Support: SQA, SCM, ILS and Test personnel
5. Development Support: development personnel that may also perform quality, configuration management, logistics and test activities during development
6. Development: system engineers and software product development personnel
7. Corporate Management: top-level corporate officers (including directors)

To meet the needs of these different categories of personnel, twelve training modules were designed. The modules are:

0. Software Process/Software Engineering for Management

1. Introduction to Process
2. Planning, Estimating, Measuring and Controlling
3. SQA/SCM/ILS/Test
4. Introduction to the Evolutionary Spiral Process (ESP)
5. Measurement
6. Real-time Structured Analysis
7. TeamWork SA/RT
8. ADARTS
9. TeamWork/Ada and Documentation
10. VAX Toolset
11. Specification and DOD-STD-2167A

A summary relating training tracks to modules is given in Table 1.

GMS uses a combination of self-paced Computer Based Instruction (CBI), self-paced Video Based Instruction (VBI), Facilitated Video Instruction (FVI)--making use of notes and a facilitator, and classroom instruction. The Computer Based Instruction available is DEC VAX-based (symbolic debugger, language-sensitive editor, etc.). The SEI Software Project Management (Continuing Education Series) video tapes are available for check out and used for self-paced instruction.

Near-term training includes development of the knowledge and understanding of software estimating and scheduling, the use of software tools that increase productivity and decrease defects, software project management, the use of Formal Inspections in the software development environment, and the changes necessary in the software development process during the transition to the use of Ada in the development process. Long-term plans have also been formulated. They identified the following needs:

- A manager responsible for implementing the training program, and a training budget
- A training group to fulfill training needs:
    - ■ Part-time instructors drawn from the GMS organization
    - ■ Consultants from external sources whose full-time job is instructor
    - ■ On-the-job training, supported by a mentor knowledgeable in the subject area
- Appropriate support tools (terminals, workstations, compilers, VAX Toolset, etc.)
- Necessary training facilities--within and external to GMS, as required
- Numerous textbooks needed to support the courses

Ms. Geri Cuthbert of the GMS Software Engineering Process Group (SEPG), and Mr. C.R. Hotz, SEPG Manager, were contacts for this summary.

Westinghouse Electric Corporation: Ms. Debbie LaPay is Manager of Software Technology and Development at Westinghouse Electric Corporation in Pittsburgh. The Pittsburgh site is engineering oriented, and had previously given little emphasis to software development practices, according to Ms. LaPay. She recognized the need to educate all personnel (managers and technical professionals) in software engineering management and practice ("to bring their understanding up to a good level"). She emphasized that this is "education" and not "training". At the same time she was faced with cut-backs in training/education funds. Their development previously made heavy use of FORTRAN; new development uses C; there is no Ada use.

The approach has been to adapt the SEI Software Project Management Continuing Education Series video tapes. She has now gone through three formal classes using the tapes, and there is a lot of individual use of the tapes. The course requires 56 hours of class time. Ms. LaPay has presented it either as one day (8 hours) each week for seven weeks, or two half-days per week for seven weeks. Both approaches have been satisfactory. Considerable progress has been made by bringing a broad spectrum of personnel together to emphasize software development. Ms. LaPay stresses the importance of having managers participate.

Raytheon Missile Systems Division: Raytheon Missile Systems Division (MSD) is located in the Boston area (Tewksbury, MA). J. Hooper heard a presentation by Forrest Gardner of MSD at

TRACK	TRAINING MODULES	HOURS OF TRAINING
1. SW Project Managers	0,1,2,3,4,5,11	40
2. Proj. Leaders/Group Leaders	0,1,2,3,4,5,6,7,8,9,10,11	148
3. Support Managers	0,1,2,3,4,5,7,9,11	128
4. Technical Support Personnel	1,3,4,6,7,8,9,10,11	140
5. Development Support Personnel	1,3,4,6,7,8,9,10,11	140
6. Development Personnel	1,2,5,6,7,8,9,10,11	176 (with Ada course)
7. Corporate Officers/Directors	0,1,2,3,5,11	26

Table 1. Summary relates training tracks to modules.

the Software Technology Conference (STC) in Salt Lake City in April 1993 [3], then communicated with Ms. Laura Acosta, the paper's author.

MSD employs 8 trainers of continuing training for about 400 software engineers. They strongly emphasize "learning by doing" in their classes. They emphasize that developing effective training programs should be approached in much the same way as developing good software: determine requirements (create syllabi), design courses, implement materials before first given, validate (i.e. dry run, etc). They have both management and engineering training tracks. With the view that the training benefits both MSD and the employees, they offer courses of 10 hours or less during working hours, and courses of greater than 10 hours half on MSD time and half after working hours.

2.3 Academia

Florida Atlantic University Graduate Education: The Computer Science and Engineering Department (CS&E) of Florida Atlantic University (FAU) in 1988 formed an Industry Advisory Committee (IAC) to assist in identifying and meeting the needs of the large computing-based industry in southeastern Florida. The members of the IAC are from Bendix King, Encore Computer Corporation, Harris, IBM, Modular Computer Systems, Motorola, Siemens Stromberg-Carlson, Racal Datacom, and United Technologies. The need for software engineering graduate education was determined to be the most urgent requirement [4].

FAU arranged to offer courses based on the SEI Academic Series video tapes at six industrial sites as follows: United Technologies (West Palm Beach), Motorola (Boynton Beach and also Plantation), IBM (Boca Raton), FAU (Boca Raton), Encore (Fort Lauderdale). The courses offered are: Software Project Management, Software Verification and Validation, Software Design, Software Creation and Maintenance, Software Specification, and Software Systems. The courses are offered for credit in FAU's Master of Computer Science degree program. A certificate program is also available.

The companies involved provided full financial support for the courses. Additional faculty were required due to teaching loads, and all had industrial software engineering experience. Each course has three semester hours credit. The classes meet twice each week--an hour for one session, and two hours for the other session. An SEI tape typically has been shown during the one-hour class, and the instructor presents additional material, interacts relative to assignments, etc., during the two-hour class, and also may present some video material.

As of Spring term 1993, over 250 students had taken at least one course, and an additional company site at Bendix King was added. They anticipated that the demand for the courses would be satisfied in the area within a couple of years, but demand remains strong. Managers surveyed expressed the expectation that the courses would "change culture" at their sites, and benefit software productivity. The students mostly volunteered to participate, and managers reported enthusiasm of student participants. The managers indicated that the attendees were their more capable personnel--although some managers indicated that only the better employees volunteered for the courses. They reported that the greatest gain to their organizations was in the requirements/design area. The managers felt they had changed their own management style and techniques, at least relative to the course participants. In one case, a steering committee was formed and meets regularly with the manager to discuss software development strategies. The managers unanimously recommended the courses to others.

Some points mentioned for future consideration include how to coordinate software engineering education with respect to management involvement, how to better integrate formal and rigorous material into software engineering education and industry use, and how to rapidly acknowledge the changes in capabilities of participating employees in order to expedite culture change; i.e., how to utilize the missionaries.

Marshall University Undergraduate Education: Marshall University (MU) in Huntington, WV is one of very few colleges/universities with what could be called an undergraduate major in software engineering. Florida Institute of Technology is another notable example of such a program. The evolution of MU's B.S. in Computer Science to a strong degree program with a pronounced software engineering emphasis is described in [5].

The program was designed with active input from the Computer Science Advisory Panel, consisting of senior computing managers from major companies in the Tri-State area. Emphasis is placed on science and mathematics, as well as courses in economics, management, and accounting. The software engineering life cycle is explained in the first freshman course, and provides focus to all computer science courses. Ada is used as the teaching language, and group participation is stressed throughout the program. At the junior level students take two three-hour courses in software engineering. At the senior level a full-year capstone team project involves the students in the entire life cycle of a "real project" of substantial size, for a "real customer".

We began phasing in this new program during 1992-93, and will offer the capstone sequence first in 1994-95. We have high expectations for the graduates of this program, as do the member companies of our Advisory Panel.

3. Identification of Required Knowledge and Skills

In this section we first document our assumptions underlying the determination of required knowledge and skills, and then present the knowledge and skills requirements identified.

3.1 Underlying Assumptions

In this subsection we present what we consider to be important presuppositions, upon which we formulate requirements for knowledge and skills in the following subsection.

Ada's role in software development and maintenance pertains primarily to the implementation phase. The emphasis on Ada within many organizations has no doubt brought about increased emphasis on the other life cycle phases, and Ada/PDL (program design language) is useful in the design phase. Implementation averages about 20 percent of development cost for large software systems, and maintenance costs exceed development costs by factors of 2 to 4, or more [6]. If the other development and maintenance activities not pertaining directly to Ada were ignored, the overall improvements needed in software productivity and quality would not be attainable. Thus:

> *Ada should be learned and used within the larger framework of an effective software engineering process.*

The Software Engineering Institute (SEI) at Carnegie Mellon University has taken the lead in the United States in emphasizing the importance of well-defined and enforced software engineering processes within organizations. It has formulated five levels of maturity against which processes may be measured. This thrust is in keeping with a strong international emphasis on quality. A great many companies have set corporate goals to improve their processes based on growing use of process maturity in contract awards. Thus it is apparent that:

> *Effective software development and maintenance require a well-defined, repeatable, enforced process, with all personnel sharing a common commitment and understanding.*

Automated tools can play an effective role in software engineering, but they are certainly no substitute for adherence to a carefully defined process. Rather, such tools should be chosen and used based on the process, including specific phase-based methodologies. Further, personnel effectiveness in software engineering comes from the consistent, repetitive application of methods, languages, and tools, with accompanying accumulation of greater understanding, insights, and skills. We understand this principle in other professions and crafts -- that success comes from experience, and that successful professionals and craftsmen do not change their practices without strong justification. In summary:

> *Specific controls, methods, and tools should be selected and enforced for use within a software engineering process.*

In most organizations, many standards and guidelines direct and constrain the creation and maintenance of software. Personnel at such organizations have wide-ranging duties relating to the development and maintenance of software. Such duties range across such activities as interacting with functional personnel to formulate software requirements based on application-area needs, performing cost estimates, interpreting acquisition regulations, contracting for software development or maintenance, conducting reviews throughout the life cycle, actually developing software in-house, assuring the quality of the resulting software, as well as many others. In summary:

> *Personnel may have many different roles in the creation and maintenance of software, including program management, acquisition, and contract management, as well as in-house development and maintenance of software, with differing needs for knowledge and skills.*

3.2 Knowledge and Skills Requirements

Based on the assumptions stated in 3.1, and the state-of-practice of effective software engineering, we offer the following formulation of "modules" of software engineering knowledge and skills, with partitioning guided by the need to identify training and education to supply the needed knowledge and skills.

M1: Introduction to Software Engineering and Software Project Management

- Software and software engineering: definitions
- The software process: definitions; quality and productivity issues; process maturity/improvement
- Life-cycle phases/activities: development and maintenance
- Life-cycle approaches: waterfall, exploratory, prototyping, incremental development, formal transformation, spiral,
- Rapid Development Paradigm
- Verification and validation
- Reliability and safety
- Software reuse (integral to the life cycle)
- Embedded applications/real-time/MIS
- Applicable standards: 2167A/7935A/SDD, Ada, ...
- CASE tools/environments
- Ada: significance and contribution
- Software project management fundamentals:

  The "business case" for a project
  Planning and scheduling
  Milestones, reviews, inspections
  Measurement, metrics
  Risk assessment/management
  Configuration management
  Cost estimation and control

Quality assurance
Human factors
Documentation

M2: Requirements Definition and Specification

- Systems engineering approaches
- Requirements modeling, prototyping, simulation
- Specification techniques and notations
- Characteristics of a "good" requirements document
- Function-oriented approach
- Object-oriented approach
- Non-functional requirements
- Reuse emphasis
- Maintainability emphasis
- Verification and validation

M3: Software Design

- The design process
- Design strategies and notations
- Function-oriented approach
- Object-oriented approach
- User interface design
- Reuse emphasis
- Maintainability emphasis
- Real-time issues
- Prototyping and simulation
- Iterative development issues
- Verification and validation

M4: Software Implementation with Ada

- The Ada language: syntax and semantics
- Detailed design (Ada/PDL)
- Code generation strategies with Ada
- Fault avoidance/tolerance
- Reuse emphasis
- Maintainability emphasis
- Code generation tools
- Iterative development issues
- Verification and validation
- Unit testing

M5: Software Verification, Validation, and Quality Assurance

- Reliability and safety
- Test planning/strategies
- Testing and debugging
- Static verification
- Support tools
- Inspections and reviews
- Metrics
- Quality assurances

M6: Application Knowledge

Although not software engineering knowledge, per se, application knowledge is indispensable in the development of a software system. Application domain specialists may be necessary as part of a team, along with systems and software engineers. It should be emphasized, however, that successful development of large software systems depends on software specialists planning and guiding the project.

It is evident on examination that module M1 encompasses at an abstract, less-detailed level the concepts included in modules M2 through M5. Many other specialized, in-depth topics could be included, such as Design of Real-Time Systems, Process Improvement, Cost Modeling, Formal Methods, etc. However, in the current context this set of "modules" should serve our purposes.

4. Curriculum Content and Implementation

Based on the survey of education/training experiences in section 2 and the knowledge and skills requirements identified in section 3, we formulate some suggestions relative to curriculum content and implementation. We consider the modules to be curriculum needs, since we must identify education/training to help bring about the knowledge and skills. We will summarize the modules here; please refer to section 3.2 for details of each module.

M1: *Introduction to Software Engineering and Software Project Management*
M2: *Requirements Definition and Specification*
M3: *Software Design*
M4: *Software Implementation with Ada*
M5: *Software Verification, Validation, and Quality Assurance*

We also listed module *M6: Application Knowledge*, which we will not directly address.

We also need to state our assumptions about the different categories of personnel who may need education/training. One of the training experiences we reported (Grumman, section 2.2) was based on seven categories of personnel. We believe it will be sufficient for our purpose to consider four separate categories of personnel within an organization.

- Software Practitioners
- First-level managers (group managers/leaders)
- Mid-level managers
- Top-level managers

It is apparent that no clear-cut dividing line exists between the third and fourth categories. However, our intention is that "top-level manager" means at least a third-level manager, and perhaps higher than that, who perhaps has responsibilities in one or more functional areas in addition to software. Further, we will understand that group leaders/managers and mid-level managers may be managing in-house or contracted software development/maintenance, or software quality assurance. Similarly, practitioners may be conducting contract oversight/monitoring activities, or SQA or other support functions, rather than or in addition to software development and maintenance. We offer suggestion of kinds and approximate hours of instruction for each personnel category as shown in Table 2.

The module M1 content is, in our opinion, critically important in order to establish a culture of strong software emphasis within an organization. As our earlier observations indicated, it is important for both practitioners and managers to have this course, and can be most effective if each course includes both practitioners and managers. M1 is also very important as the foundation for all the other modules. Even personnel with undergraduate degrees in computer science should probably take M1 along with everyone else: they very likely had little software emphasis in their curriculum, and if they did, and/or if they have had an excellent background of experience in software engineering, they can make a good contribution in class discussions and group laboratory projects.

Available sources for this education and training include commercial companies, SEI/CMU, universities/colleges, and in-house custom courses. Some companies offering a variety of software engineering and Ada courses are FASTRAK Training, Inc., EVB Software Engineering, Inc., and TEXEL & COMPANY, Inc. The commercial companies may develop custom courses, or adapt their standard courses. Also numerous workshops, seminars, and short courses are offered throughout the year by many companies and individuals, and are widely advertised. Moreover, some computer-based tools are available for Ada instruction and reinforcement. They are not adequate to replace training courses, but could have some benefit in conjunction with such courses. In the report [7] we identified specific alternative sources for the courses, but will not include that material here due to space constraints.

We have previously stated our opinion that essentially all software-related personnel should take an M1 course, preferably including combinations of practitioners and managers in each course offering, and that it should be taken prior to taking any one of courses M2 through M5. We believe that the most effective way to provide such a course would be to approach it from the standpoint of software process, as NRaD has done; i.e., define the process, then devise an M1 course based on the process. Developing a custom course would permit inclusion of organizations-specific process aspects. A disadvantage of this approach is the time and initial cost needed to develop and field the course, as compared with making use of existing vendor courses. However, once the course is developed, it could be used repeatedly and widely with a fairly low level of effort needed to keep it updated. Such a course could make use of some SEI material, for example, as is done by Westinghouse, which would reduce time and cost to develop the custom course.

Another approach is to rely on vendor courses (off-the-shelf or customized). This has the initial advantage of course and instructor availability. Over the longer term it is likely to cost more than the in-house custom approach, and is unlikely to satisfy specific needs as well as a custom course. Due to the need for interaction and team-building during the M1

MODULE	APPROX. HRS. OF INSTRUCTION			
	Practice	1st-Level	Mid-Level	Top-Level
M1: Intro. to Soft. Eng. and Soft. Project Management	40	40	40	>=16
M2: Requirements Def. and Spec.	40	40	>=8	>=0
M3: Software Design	40	40	>=8	>=0
M4: Software Impl. with Ada	80 *	80 *	>=16	>=8
M5: Verif./Valid./SQA	40 **	40 **	>=8 **	>=0

* An introductory programming course may be necessary prior to Ada training for personnel without programming experience. In this case the introductory programming course perhaps should be taken prior to taking the M1 course.

** Is needed especially by SQA personnel.

Table 2. Suggestion of kinds and approximate hours of instruction for each personnel category.

offerings, we believe that providing M1 courses on-site at the organization facilities would be considerably superior to distance-learning M1 courses. A possible alternative would be to employ video teleconferencing.

Outside consultants could be used to help develop a custom course, or to audit and assess the quality of available vendor courses. An organization may pursue a process-related education/training approach, and undertake development of a custom M1 course, using consultants as needed. Simultaneously some vendor courses could be offered if the need is too pressing to wait until a custom course is available. Lessons-learned from these courses could provide some benefits in developing the custom course. Whatever approach is taken, it will be important to have a shorter version of the course for top-level managers.

In our opinion the source of offerings for M2 through M5 is somewhat less critical than for M1. However, we believe that these courses as well as M1 should be designed/chosen based on process considerations. M2 should reflect a balance between functional decomposition and object orientation, with greater emphasis on the organization's desired future methodology. The initial cost and time investment to develop a custom M1 course applies to M2 through M5 as well (an initial comparative disadvantage relative to vendor courses). However, similarly to M1, custom courses would have longer-term cost benefits, as well as serving the need for education and training more precisely. A distance-learning format could be acceptable for offering these courses (in contrast to our comments about M1 above).

Module M4 (Ada) could more appropriately be called "training" than the other modules. M4 has the least criticality relative to the source of the courses, compared to the other modules. The computer-based Ada tools leave a good deal to be desired currently. However, they could have some merit for use in conjunction with formal instruction.

Various short courses offered by individuals and companies have a place in an organization's program of education and training. For example, currently offered courses in MIL-STD-SDD and the Yourdon-Coad approach to object-oriented design could be useful to augment M1 through M5 courses, once individuals are adequately grounded in fundamentals. TQM and other quality-related courses are other examples of appropriate courses to provide later.

Many professionals desire to have academic credit for software engineering education, and these opportunities should be provided to some personnel. This can ensure a greater depth of expertise within organizations. However, academic education cannot replace in-house and continuing education course offerings, since not all software personnel could, or would, take the academic courses. Academic offerings can be obtained in distance-learning format, as well as from nearby campuses, and occasionally through special arrangements in which professors teach on-site courses.

5. Conclusion

Reviewing education and training programs offered by broad spectrum of organizations has been helpful to us in formulating a broadly-based set of suggestions. Our emphasis has been to offer practical suggestions for use by organizations that are directly involved with software management, development, and maintenance -- especially those using or planning to use the Ada language.

We proposed a fundamental curriculum and discussed alternatives for implementing it. Previous successes of numerous organizations in upgrading the skills and knowledge of their personnel should provide encouragement to other organizations contemplating such efforts.

Acknowledgment:

This work was sponsored by the US Army Corps of Engineers under Contract No. DACW69-90-0012, Mitchell Systems Corporation, Work Order No. MSC3-024. The content of this paper is a summary and adaptation of the report prepared under the Work Order [7]. We express appreciation to the companies and individuals providing input to our survey.

References:

[1] Hayward, Duston. "Software Process Improvement Through Training." Software Engineering Process, Office, Naval Command Control, and Ocean Surveillance Center, San Diego, CA, 1993.

[2] Smythe, Walter E., "Government-Academic Partnership for Software Engineering Education." Proceedings of the Software Technology Conference, Salt Lake City, UT, April 1993.

[3] Acosta, Laura. "How to provide Quality Software Training." Proceedings of the Software Technology Conference, Salt Lake City, UT, April 1993.

[4] Coulter, Neal and James E. Dammann. "Changing Culture Through Software Engineering Education." Computer Science and Engineering Department, Florida Atlantic University, Boca Raton, FL, 1993.

[5] Hooper, James W. "Planning for Software Engineering Education within a Computer Science Framework At Marshall University." Proceedings of the Software Engineering Education Conference, San Diego, CA, October 1992, pp. 257-269.

[6] Sommerville, Ian. Software Engineering, 4th Ed., Addison-Wesley, 1992.

[7] Hooper, James W., Akhtar Lodgher, and Hisham Al-Haddad. "Ada Migration Training Plan for the U.S. Army Corps of Engineers." USACE Contract DACW69-90-0012, Work Order No. MSC3-024, Marshall University, Huntington, WV, June 1993.

THE PRODUCT-DRIVEN DEVELOPMENT OF A REAL-TIME ADA EXECUTIVE FOR THE SPACE SHUTTLE COCKPIT UPGRADE

Charles A. Meyer, Gary H. Barber, David Soto

Intermetrics, Inc.
1100 Hercules Drive, Suite 300
Houston, Texas 77058

Abstract

This paper describes an eclectic approach to software development that emphasizes orientation of effort toward the end product: delivered code. The approach describes both project management and development techniques that minimize deterrents to a high quality, low cost product completion. The core of this approach uses a compilable Ada PDL and the semi-automated generation of selected documents. It also places a heavy emphasis on prototyping and iterative development; this, along with the use of a compilable Ada PDL, places risk reduction and quality improvement at the center of the method. Finally, product-driven development encourages the careful consideration of each required document's content. Careless documentation approaches divert attention from the *software* as the product (rather than the *documentation* – or the design methodology). The approach is being successfully applied to development of an embedded real-time executive for the Space Shuttle.

Introduction

If we were to summarize the intent of product-driven development, it would be simply "Deliver the best possible product with the least amount of effort." Although that may sound trite, development efforts encounter many opportunities to fall short of this goal. Intermetrics has refined a product-driven approach that places the emphasis on developing the delivered code, automating development of documentation, and minimizing risks.

Product-driven development encourages the use of any technique that reduces effort and improves quality. Effective product-driven development, like Total Quality Management, requires a continuous evaluation of the process in place, with an eye towards process improvement.

The basics of product-driven development can best be understood in the context of an effort vector. The effort vector expresses the amount of effort expended towards product completion and the deflections of effort away from that goal.

The Effort Vector

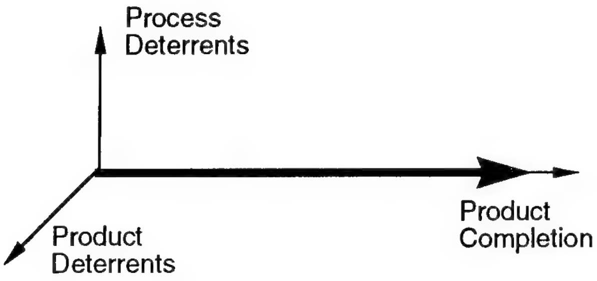
Given the start of a software development project, one can imagine effort as a distance that must be covered to complete the implementation. This distance is minimized when nothing interferes with the completion of the project. It is increased by two kinds of deterrents: process deterrents and product deterrents.

Process deterrents include inefficient or ineffective communication, coordination, and change management. Examples include baselining a volatile preliminary design document, or rewriting design documents to satisfy new interpretations of the required format or contents.

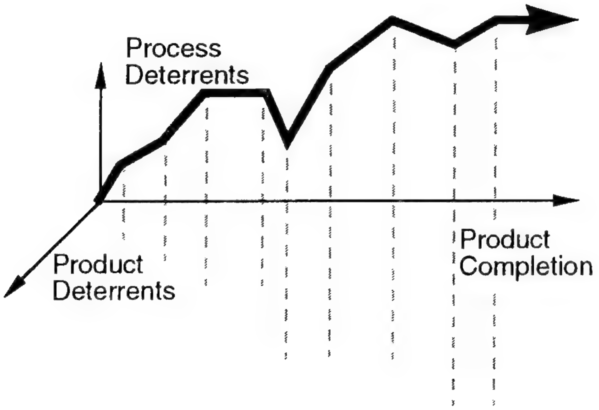
Product deterrents include the use of inefficient or ineffective development tools, the need for extensive corrective action to a product, a poor experience base with the development environment or application, and an inefficient documentation approach. Examples include using CASE tools when doing so requires an inordinate amount of effort to use the tool, or selecting a documentation approach that requires six-to-ten pages of documentation per page of code.

The distance – including the effects of deterrents – can be expressed as an effort vector. The magnitude of the vector represents the effort expended to complete a project. The effort vector for a project that experiences no deterrents is shown in Figure 1a. Figure 1b shows a more typical effort vector for completing a product.

The goal of product-driven development is to aggressively reduce the deflections due to deterrents, or, stated another way, to minimize the distance from project initiation to product completion.



a) Idealized effort to complete a project: All project efforts coherently contribute to the successful completion of a product; no deterrents interfere.



b) A typical project: Overhead is introduced as the result of deterrents; the goal is to minimize such deflections.

Figure 1. Impact of Deterrents on Effort

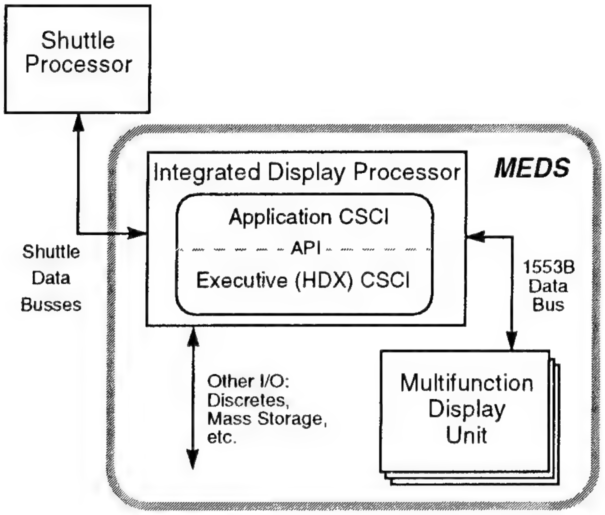
This mindset of "developing the best possible product with the least amount of effort" goes hand-in-hand with the use of modern software development techniques such as iterative or spiral development. The key, however, is not in the use of a specific technique but in continuously and conscientiously seeking out ways to reduce or eliminate deterrents while improving quality.

The remainder of this paper discusses how Product-Driven development was applied to the development of a real-time executive for the Space Shuttle.

The MEDS Project

The Multifunction Electronic Display Subsystem (MEDS) project is a NASA project to upgrade the Space Shuttle cockpit instrumentation. In this two-phased project, the current cathode ray tube displays, the navigation and flight control instruments, and several analog indicators will be replaced by multi-function flat panel displays. An abbreviated configuration of the MEDS architecture is depicted in Figure 2.

Intermetrics is developing an executive that runs as one of two Computer Software Configuration Items (CSCI) resident in the Integrated Display Processor (IDP). The IDP provides the interface between the existing Shuttle processors and the new display processors. The other CSCI is the application software, which uses data from the Shuttle processor to generate the graphic objects transmitted to the Multifunction Display Units (MDU).



Shuttle Processor
Integrated Display Processor
MEDS
Application CSCI
API
Executive (HDX) CSCI
Shuttle Data Busses
1553B Data Bus
Other I/O: Discretes, Mass Storage, etc.
Multifunction Display Unit

Figure 2. Abbreviated MEDS Project Architecture

Two CSCIs in the same processor developed by separate contractors resulted in a unique software-to-software interface.

A tailored subset of the DoD-STD-2167A Data Item Descriptions (DID) defined the deliverable documentation suite for the software.

The Executive

The Hardware Dependent Executive (HDX) is designed to manage the hardware and run-time resources for the application software in the IDP. The executive provides I/O and general purpose services. I/O services include DOS compatible file management for EEPROM and a SCSI hard drive as well as Shuttle and MIL-STD-1553B data bus, keyboard, and switch services. General purpose services include program management, self test, and interval timer services. The Ada run-time services are provided by a commercial off-the-shelf (COTS) run-time environment.

The executive presents an Application Program Interface (API) to the application running in the IDP. The API provides service calls, event notification, exception handling, and access to shared data.

Applying Product-Driven Development

The product-driven approach used to develop the MEDS executive always focused on the software as the (primary) end product and the documentation as a (secondary) by-product. As the following sections will show, this focus was maintained by

- selecting appropriate risk management techniques
- designing the executive using PDL[1] that evolved to source code
- developing custom software tools to automatically generate large portions of design documents
- applying supportive project management techniques
- scrutinizing DoD-STD-2167A documentation requirements (the DIDs) to find a best fit between the conflicting concerns of document quality and usability, and the reduction of effort required to produce them.

Risk Management

We applied techniques for reducing technical risks at every opportunity. Rather than relying on traditional paper analysis, design walkthroughs, and code inspections to uncover errors, we selected techniques that provided direct, unequivocal feedback. This included using iterative development[2], prototyping, benchmarking, and compilable Ada PDL. Traditional approaches were useful as supplements in isolated, low-risk situations. Table 1 provides a summary of some risks identified during the project and the mitigation strategy employed.

Use of a Compilable PDL

Perhaps the decision that had the largest impact on our project was to use compilable Ada PDL as the primary vehicle for developing the executive. This allowed us to reduce the distance to the completion of a high-quality product in several important ways. The developers made more effective use of Ada, the implementation was much easier to complete, and the design documents were easily generated from the PDL.

Table 1: Key Risks and Mitigation Strategies

Risk Category	Risk Items, [Related Deterrent][a]	Solution Employed	Discarded Alternatives
External Interface	Communicating the API's functionality *[Process, Product]*	Well-commented Ada package specifications; Early deployment; Frequent updates	Following standard 2167A IRS/ IDD documentation
Design	Major Architectural Decisions *[Product]*	Prototyping	Paper analysis
	Target Hardware Unavailable until After CDR *[Process]*	Iterative development on surrogate target	Hardware simulation; Use of walkthroughs as primary means of corrective action
	Low experience level with system programming in Ada (e.g., LRM Chapter 13) *[Product]*	Compilable PDL; Prototyping; Iterative development; Brief tutorials on key topics	Reliance on independent study and design walkthroughs as primary means of corrective action
Performance	RTE Performance *[Product]*	ISR and tasking benchmarks	Paper analysis based on PIWG benchmarks
	Service Efficiency *[Product]*	Benchmarks; Examination of compiler-generated assembly language	Paper analysis based on estimated CPU performance
Schedule	Documentation Overhead *[Process]*	Automatic document generation; Careful DID interpretation; Single point of change	Commercial CASE tools

a. *Process* = Related to Process Deterrents
Product = Related to Product Deterrents

Early Benefits of PDL. Developing compilable Ada PDL early on in a project allows the designers to work in the medium in which code will be written: Ada. Traditional design approaches emphasize the use of an intermediate representation – text, diagrams, or formal notations. This often divorces the designer from the realities of the implementation. The result can be a tremendous amount of time spent looking for "reality disconnects" in the design. Or, the penalties may not be accrued until coding, when such flaws become obvious (requiring design updates and further reviews).

On the other hand, when designers use compilable PDL, many implementation details can be checked by a standard Ada compiler (e.g., Ada syntax, unit interfaces, etc.) This orientation towards implementation details pervades the design thought process.

Thus, difficulties mapping the design to an implementation or correctly using Ada are largely avoided, improving quality (and reducing deterrents).

Implementing the PDL. Depending on the degree of detail present in the PDL (often a function of available time), implementation is usually a short step. The designer refines generalized algorithms by replacing or augmenting high-level descriptions with calculations and subprogram calls. Also, the designer may add new support units and control flow constructs. There is no shift in focus or activities in the transition from design to implementation (which helps reduce process deterrents). Compilable PDL becomes executable code.

Single Point of Change. The ability to equate compilable PDL with executable code was a key benefit of our approach. The PDL is not discarded when coding begins; rather, executable statements are added to the PDL until it is fully functional. Therefore, the potential exists for a single point of change for the design AND the source code. We capitalized on this by developing a small set of software tools that take Ada source code/PDL and generate design documentation. This means that design changes are made to one item: the source/PDL. Design documents are re-generated by running a collection of tools that operate on the source/PDL. Thus, in the latter stages of the project, we were essentially reverse-engineering the source code into it's equivalent PDL design. Any change to the design or implementation was automatically reflected in the documents by execution of the tools. This clearly reduces process deterrents.

PDL and Source Code as Design Documentation. Our first observation is that Ada is an extremely readable language. This allows the direct use of Ada to express a design; this would probably not be an effective approach in most other programming languages. The use of Ada also eliminates the ambiguities that cloud designs expressed using only structured English.

Also, by carefully selecting a small set of coding and commenting conventions, both the PDL and the resulting source code can be easily processed by a small collection of tools to automatically generate detailed design documentation. Care was taken to avoid PDL-specific constructs that cluttered or obscured the source code (as we've seen with some other approaches). See Attachments A and B for more details.

Document Generation

One of the major effort saving aspects of our approach was the semi-automatic generation of design documentation directly from the PDL. This was accomplished by a set of custom and commercial software tools. These tools would parse Ada source files to extract design information, format PDL, generate diagrams, and organize the results in accordance with 2167A format requirements.

These tools generated the bulk of two design documents: the interface specification for the API, and all of the detailed design in the executive's design document. Figure 3 illustrates the automated approach used to generate the documentation. Attachment C provides samples of the tool's output for an Ada package.

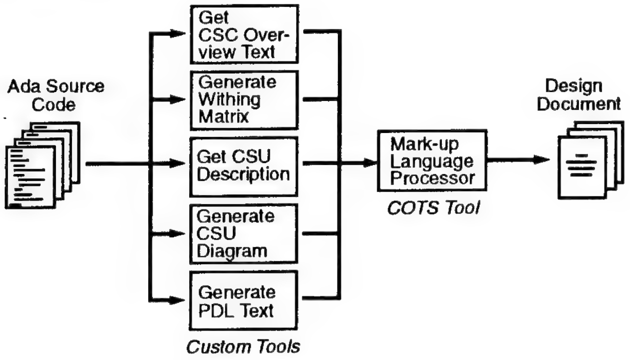


Figure 3. Automated Documentation Approach

An important aspect of the MEDS product-driven approach consisted of using custom and COTS software to automate the generation of the documentation from Ada source code.

Custom Software Tools. We developed custom software – the Intermetrics' Maker/Ada Design Generator (IMADGen) tools[3] – to support our implementation of the product-driven process. However, we always considered commercially available CASE tools first. After evaluating them for suitability to a low-overhead, iterative development approach, we usually chose to build our own tools. Often, COTS CASE tools were too constraining or had too high an overhead (both in learning curves and usability) to fit into our desired approach.

Some of the custom MEDS tools originated from a previous project[4] in a PC/DOS environment. These tools served as a starting point, but we improved them in the process of porting them to a new environment. Their availability was a motivating factor in the decision to develop custom tools.

We used whatever means were available to expedite tool development to keep costs down. Working in the Unix environment was a big advantage because of the excellent tools available for text processing and the sophistication of its scripting language. We were able to further reduce effort by following two principles in developing a new tool:

- start with a bare-minimum capability and let subsequent use of the tool suggest incremental enhancements
- adopt the Unix philosophy of building new tools by stringing together existing ones.

Unix commands (e.g., sed, awk, grep) and shell scripts were heavily used when a quick solution was desired. This was often the most direct approach when text processing was involved, due to Unix's strength in that area.

Although some of the tools were written in Ada, some were written in C. For example, one C program generates markup language drawing commands for a Booch-style diagram[5] of a package's exported types, constants, exceptions, and subprograms. Another C program generates drawing commands for a withing relationship matrix among collections of packages.

The desktop publisher (the COTS tool in Figure 3) provided a text-based markup language that allowed the tools to generate commands to format text, number paragraphs, figures, and tables, as well as draw figures for package specifications. The tools used were all tailored to generate markup language output so it could be directly imported into the design document without any manual formatting.

Project Management

Several project management methods were employed that contributed nicely to our product-driven approach. These included the use of proof-of-concept techniques (Iterative Development, prototyping and benchmarking) and brainstorming sessions.

Use of Iterative Development.[6] The availability of the target platform late in the project hampered our ability to iteratively develop the executive. However, we configured an 80386-based PC to closely match some key hardware capabilities of the target. This allowed us to develop several capabilities (DOS-compatible file I/O to a SCSI hard drive, 1553B Bus Controller I/O, interrupt handling, and ISR-to-task event posting) well before target hardware was available. In addition, this provided developers with hands-on experience with the embedded cross-compiler's toolset and RTE.

Use of Prototyping and Benchmarking. Limited prototyping allowed core design decisions to be explored. Extensive benchmarking was performed to evaluate the behavior of the Ada run-time environment. The benchmark results caused executive, client application, and system design issues to be revisited, which resulted in design changes. Also, prototyping was used early in the design process to allow developers to gain direct experience with critical language, run-time, and hardware interface issues.

Use of Brainstorming Sessions. Team members frequently conducted brainstorming sessions throughout the project. These sessions were invaluable in evaluating and selecting alternatives for key issues. These issues often involved how to most effectively approach a new task. In other cases, we brainstormed on how to refine existing activities or products. Thus, we were able to informally satisfy the Total Quality Management's process improvement cycle of plan, implement, evaluate, and modify. We note that there is a certain chemistry required for brainstorming to work well; we were fortunate to have had that chemistry.

Interpreting 2167A DIDs

DoD-STD-2167A development standards can be a useful, manageable guide to the development process. However, the burden is on the developer to be creative in taking advantage of the flexibility 2167A provides[7]. This assumes, of course, that the project sponsor is flexible and willing to consider modern (and sometimes unconventional) applications of the standard. For example, even though 2167A explicitly states it wasn't intended to impose a particular development method, there is nonetheless often much concern and debate when a non-waterfall model is employed.

CSC and CSU Mappings. Perhaps the single biggest decision that affects the amount of documentation (increasing effort while contributing little or nothing to quality) is the selection of a mapping of Computer Software Components (CSCs) and

Computer Software Units (CSUs) to physical Ada software items[7,8,9]. These terms are critical because design documents must provide specific preliminary and detailed design information for CSCs and CSUs, respectively. The more primitive the mapping, the more documentation is required for the same implementation.

For example, two mappings are most common: 1) CSCs = Ada library units, CSUs = procedures, functions, or tasks; 2) CSCs = an abstract architectural entity (similar to Booch's subsystems[10]), CSUs = Ada library units. The first mapping easily doubles or triples the bulk of the design documentation and results in a maintenance nightmare. Preliminary design must be updated whenever a library unit (CSC) is added or deleted. Detailed design must be updated whenever a procedure, function, or task (CSU) is added or deleted (or renamed). In addition, test documents must trace test cases and requirements to the CSU level. If a project consisted of 200 library units, each of which contained 5 to 20 subprograms, then 1,000 to 4,000 CSUs would need to be explicitly documented, maintained, and traced to. Clearly, this mapping results in documentation requirements driving the design process, rather than the design[7]. The use of the second mapping, however, would result in a much more manageable 200 CSUs, with no loss of visibility into the design.

Inadequate Requirements for CSCI-to-CSCI Documentation. One final note on 2167A is its inadequacy for documenting software-to-software procedural interfaces. The Interface Requirement Specification and Interface Design Document (IDD) are heavily slanted towards specifying message-based interfaces involving communication channels. This is an awkward slant when attempting to document an API. In fact, we found we could not defend any interpretation of the IDD DID for the API. Instead, with the concurrence of our customer, we defined our own format and content requirements and called it an Interface Requirements and Design Document (IRDD). The API IRDD was composed mostly of Ada package specifications, which nicely expressed the API's design and its test-to requirements.

Summary

By using the strategies outlined in this paper to maintain focus on the delivered software, we are accomplishing the MEDS executive project ahead of schedule and under budget. The approach presented in this paper resulted in high-quality 2167A documentation and high-quality Ada source code with less effort than any other approach these authors are familiar with. It also has the added benefit of working well with virtually any software process model or design methodology. It was well received by our customer, and was even adopted by them midway through the project.

Product-driven development defines a seamless design and implementation process with easy evolution through prototypes and delivered code. Early prototyping and benchmarking are key to avoiding or mitigating technical and schedule risks.

By employing these techniques, we were able to verify critical portions of the design early in the project. In the process, the designers had an opportunity to gain experience with the idiosyncracies of the hardware and software target environment. In addition, our development of simple tools to generate design documentation from compilable Ada PDL made a major contribution to the reduction of development effort (and easily paid for itself).

In summary, product-driven development can be considered a "back to the basics" approach with the aim of putting the major emphasis on the real desired product - the executing code.

Acknowledgments

We would like to thank Al Bishop and Rick Weyrauch of Southwest Research Institute (San Antonio, TX) for early contributions to the use of compilable Ada PDL and document generation. We would also like to thank John Fumo of Honeywell Satellite Systems Organization (Phoenix, AZ), for his support and feedback in applying the product-driven approach to the MEDS project.

Bibliography

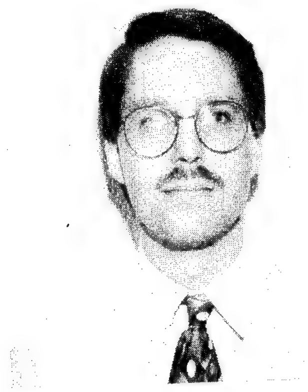
1. Ken Shumate, Marilyn Keller, *Software Specification and Design, A Discipline Approach for Real-Time Systems*, John Wiley & Sons, Inc., New York, NY, 1992, pp. 254-255.
2. Richard Simonian, "Software Development in Core: The Application of Ada and Spiral Development," TRI-Ada '92 Conference Proceedings.
3. Intermetrics' Maker/Ada Design Generator (IMADGEN) Tools User's Guide, an internal Intermetrics publication, 1994.
4. Alfred Bishop, "Case Study of a First Ada Train-

ing Project: The USMC Combined Arms Staff Trainer," 14th Interservice/Industry Training Systems and Education Conference Proceedings, 1992.

5. Grady Booch, *Software Engineering with Ada*, Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1983.
6. Charles A. Meyer, "The Iterative Development Method: A Means of Reducing Risk in the Development of Training System Software," 14th Interservice/Industry Training Systems and Education Conference Proceedings, 1992.
7. John A. Henry, "AdaTRAN as a Teaching Tool," TRI-Ada '91 Conference Proceedings.
8. C. Meyer, S. Lindholm, J. Jensen, "Experiences in Preparing a DoD-STD-2167A Software Design Document for an Ada Project," Tri-Ada '89 Conference Proceedings.
9. William H. Roetzheim, *Developing Software to Government Standards*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991, p. 290.
10. Grady Booch, *Software Components with Ada: Structures, Tools, and Subsystems*, Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.
11. ANSI/IEEE Std 990-1987, "IEEE Recommended Practice for Ada As a Program Design Language," The Institute of Electrical and Electronics Engineers, Inc., 1987.

Biographical Sketches

Gary Barber is the project manager for the MEDS executive software. He has participated in the development of real-time software for both commercial and government projects. These include downhole well-logging data acquisition systems and flight simulation systems. He received a BSEE from the University of Wisconsin and an MSEE from the Air Force Institute of Technology.



Charles Meyer is the lead software engineer for the MEDS executive. He has twelve years of experience developing embedded systems for process control, communications, and avionics subsystems. He has six years of experience developing Ada systems. Chuck's interests include C++, Ada-9X, GUI programming, and development processes. He received his B.S. in Electrical Engineering from the University of Wisconsin.

David Soto is a University of Texas aerospace engineer. He has been working in the NASA environment as a software engineer for more than seven years. He has experience in flight simulation; guidance, navigation, and control (GN&C); and embedded real-time systems. David is a software engineer on the MEDS project.

Attachment A: The PDL Stripper

The PDL Stripper is the tool used to convert compilable PDL or Ada source code to PDL suitable for design documents. It is compliant with the "IEEE Recommended Practice for Ada As a Program Design Language.[11]" It supports the use of both structured and unstructured comments as defined in the IEEE standard. The PDL Stripper recognizes several directives (in the form of structured comments) that affect what information in the source is passed through to the final documentation. These PDL directives are inserted as comments in the source code. The directives take the form

```
--pdl <mode>
```

The valid values for the <mode> field are:

- *Detailed:* The default configuration where only selected Ada source is passed through. Statements involving structure (begin/end, subprogram declarations/definitions, etc.), control flow, and tasking are passed through, as well as marker, documentation, and processing description comments.
- *Verbatim:* All source code is passed through unchanged.
- *Brief:* Only marker, documentation, and processing description comments are passed through; no Ada statements are present in the output.
- *Suppress:* Nothing is passed through. This is most useful for deleting whole sections of code/PDL, such as machine code insertions.
- *Substitute:* Takes comments and turns them into PDL. For example, Figure 4a becomes Figure 4b. This is useful for providing PDL that's totally independent of the source code, as in the case of machine code insertions.

```
--! if A < B then
--!   --[ Process input.
--! else
--!   --[ Transmit a message.
--! end if;
```

a

```
if A < B then
  --[ Process input.
else
  --[ Transmit a message.
end if;
```

b

Figure 4. Converting Comments to PDL in Substitute Mode.

More information on the PDL Stripper's additional structured comments, reserved sentinel characters, and unstructured comments are defined in the IMADGEN Tools User's Guide[3].

Attachment B: The PDL Package

A PDL package was developed to allow designers to write compilable Ada designs without forcing early design decisions. This package (shown below) provides predefined procedures, types, and objects to facilitate such deferrals.

```
package Pdl is
   -------------------------------------------------------------
   --| Description:
   --|   This package contains declarations that are useful for
   --|   designing Ada software using program design language
   --|   (PDL). Given the constructs in this package, a designer
   --|   is able to create compilable PDL. This ability to compile
   --|   allows the compiler to be used as an interface checker
   --|   between units, as well as a limited design checker.
   --|
   --|   The elements provided include constant data elements,
   --|   special types, and a procedure. The recommended use for
   --|   these elements are documented where they are defined.
   --|
   --|   Note that applications developed using this package can
   --|   gracefully be transitioned to executing code as PDL
   --|   statements are gradually replaced with real source.
   --|
   --| Constraints & Limitations: None.
   --| Initialization Exceptions: None.
   -------------------------------------------------------------

   Unknown_Int_Value   : constant := -1;
   Unknown_Float_Value : constant := -1.1;
   --| Named numbers for identifying constants whose values are not
   --| yet known.

   type Tbd is range -1 .. 10;
   --| Use this to give names to types whose structure is not yet
   --| known, e.g.
   --|   type Message_Buffer_Status is new Tbd;

   type Choice is (Option1, Option2, Option3, Option4, Option5,
                   Option6, Option7, Option8, Option9, Option10);
   A_Choice : Choice;
   --| Use these for case statements, e.g.
   --|   case A_Choice is
   --|      when Option1 => -- describe the option here
   --|         ...
   --|      when Option2 => -- etc.

   Condition : constant Boolean := False;
   --| Use this to create PDL control structures, e.g.
   --|   if Condition then ...

   List_Of_Items : constant array (Choice) of Tbd :=
                  (others => Unknown_Int_Value);
   --| Use this to create "for" and "while" loops, e.g.
   --|   for I in List_Of_Items'Range loop
   --|      --[ Process items.
   --|      Statement;
   --|      --condition: a special item was found
   --|      exit when Condition;
   --|   end loop;

  procedure Statement;
   --| Use this wherever an Ada statement is required, e.g.
   --|   if Condition then
   --|      --[ Perform processing for "True".
   --|      Statement;
   --|    else
   --|      --[ Perform processing for "False".
   --|      Statement;
   --|    end if;
   --| Note that this is safer than using "null", since removing
   --| the "with Pdl;" context clause when coding is complete
   --| guarantees there are no overlooked areas, since "Statement"
   --| will no longer be available and the source will not compile
   --| without errors.

end Pdl;
-----------------------------------------------------------------

package body Pdl is
   --| Provide a body so that applications using package PDL will
   --| be able to link - and maybe run!
   procedure Statement is
   begin
      null;
   end Statement;
end Pdl;
```

Attachment C: Sample PDL

This section illustrates an example of compilable PDL and the corresponding design documentation produced by the IMADDGEN tools. The PDL includes a package specification and body with the proper format and syntax to generate the design document section for this CSU. Following the source is the actual output generated by the tools. The style of the output was tailored to satisfy 2167A.

```
------------------------------------------------------------------
with Api_Exceptions,
     Intel_Unsigned;

package Api_Timer is
   ---------------------------------------------------------------
   --| Description:
   --|   This package provides interval timing services to the
   --|   client. Two hardware timers can be programmed:
   --o      o Timer A
   --o      o Timer B
   --|   The state transition diagram below shows the relationship
   --|   between these services, a timer's state, and error
   --|   conditions:
   --x
   --|                         Start
   --|           +------------------------------------+
   --|     Stop  |                                    |
   --|     +--+  |            +----+ Set or           |
   --|     |  |  |            |    | Stop             |
   --|     |  v  |            |    v                  v
   --|   +----------+      +----------+          +----------+
   --|   |          | Set  |          | Start or |          |Start
   --|   | Uninit-  |----->|  Idle    |--------->|  Active  |---+
   --|   | ialized  |      |          | Restart  |          |   |
   --|   | (e.g.    |      |          |          |          |   |
   --|   | powerup) |      |          |          |          |   |
   --|   |          |      |          |          |          |   |
   --|   +----------+      +----------+          +----------+   |
   --|        |                  ^                    |         |
   --|        | Start or        .|           Stop     |         |
   --|        | Restart          |        or Timeout  |         |
   --|        v                  +--------------------+         |
   --|   +----------+                                           |
   --|   |          |                                           |
   --|   |  Error   |                                           |
   --|   |          |<------------------------------------------+
   --|   |          |
   --|   |          |
   --|   |          |
   --|   +----------+
   --x
   --| Constraints & Limitations: None.
   --| Initialization Exceptions: None.
   --| Version: 1.11  10/18/93
   ---------------------------------------------------------------
   Unknown_Fault : exception renames Api_Exceptions.Unknown_Fault;
   Status_Error  : exception renames Api_Exceptions.Status_Error;
   Use_Error     : exception renames Api_Exceptions.Use_Error;

   Min_Timer_Count : constant := 100; -- microseconds
   Max_Timer_Count : constant := 16#FFFF_FFFF#; -- microseconds

   type Kind is (Timer_A, Timer_B);
   type Mode is (One_Shot, Continuous);
   type Count is new Intel_Unsigned.Integer32
      range 0 .. Max_Timer_Count;
   subtype Interval is Count range Min_Timer_Count .. Count'LAST;
   type Status (Is_Configured : Boolean := False) is
      record
         Is_Active : Boolean;
         Was_Reset : Boolean;
         case Is_Configured is
            when True =>
               Current_Mode     : Mode;
               Current_Interval : Interval;
            when False =>
               null;
         end case;
      end record;

   procedure Set (The_Timer        : in Kind;
                  To_The_Mode      : in Mode;
                  And_The_Interval : in Interval);
   ---------------------------------------------------------------
   --| Purpose:
   --|   Sets the mode and interval of a timer for the next call
   --|   to Restart.
   --| Exceptions:
   --|   Unknown_Fault - when the service traps an unexpected
   --|                   exception.
   ---------------------------------------------------------------

   procedure Stop (The_Timer  : in Kind;
                   Send_Pulse : in Boolean := False);
   ---------------------------------------------------------------
   --| Purpose:
   --|   Stops an active timer.
   --| Exceptions:
   --|   Unknown_Fault - when the service traps an unexpected
   --|                   exception.
   ---------------------------------------------------------------

   function Status_Of (The_Timer : in Kind) return Status;
   ---------------------------------------------------------------
   --| Purpose:
   --|   Gets the current status of The_Timer.
   --| Exceptions:
   --|   Unknown_Fault - when the service traps an unexpected
   --|                   exception.
   ---------------------------------------------------------------

   procedure Start (The_Timer        .: in Kind;
                    With_The_Mode     : in Mode;
                    And_The_Interval  : in Interval);
   ---------------------------------------------------------------
   --| Purpose:
   --|   Starts The_Timer immediately after setting its mode and
   --|   interval.
   --| Exceptions:
   --|   Status_Error - when the timer is already active.
   --|   Unknown_Fault - when the service traps an unexpected
   --|                   exception.
   ---------------------------------------------------------------

   procedure Wait_For (The_Timer : in Kind);
   ---------------------------------------------------------------
   --| Purpose:
   --|   Suspends the caller until the specified timer reaches
   --|   terminal count.
   --| Exceptions:
   --|   Unknown_Fault - when the service traps an unexpected
   --|                   exception.
   ---------------------------------------------------------------
end Api_Timer;
```

```
------------------------------------------------------------------
with Hdx_Interval_Timer;

with Pdl;
use Pdl;

package body Api_Timer is

   package Hit renames Hdx_Interval_Timer;

   -- | State data
   The_Timer_Status  : array (Kind) of Status;

   ---------------------------------------------------------------
   --| Visible units
   ---------------------------------------------------------------
   procedure Set (The_Timer        : in Kind;
                  To_The_Mode      : in Mode;
                  And_The_Interval : in Interval) is
   begin
      --[ Save the configuration information for the timer.
      Statement;
      case The_Timer is
         when Timer_A =>
            --[ Write the interval to the Timer A modulo register.
            Statement;
         when Timer_B =>
            --[ Write the interval to the Timer B modulo register.
            Statement;
      end case;
   exception
      when others =>
         --[ Report the occurrence of an unknown fault.
         raise Unknown_Fault;
   end Set;

   ---------------------------------------------------------------
   procedure Stop (The_Timer  : in Kind;
                   Send_Pulse : in Boolean := False) is
      A_Choice : Choice;
   begin
      if Condition then -- Timer_A is to be stopped
         --[ Turn Timer A off.
         if Condition then -- send pulse is True
            --[ Place an event in the timer's ISR message queue.
            Statement;
         end if;
      else -- Stop Timer_B
         if Condition then -- send pulse is True
            --[ Place an event in the timer's ISR message queue.
            Statement;
         end if;
      end if;
   exception
      when others =>
         --[ Report the occurrence of an unknown fault.
         raise Unknown_Fault;
   end Stop;

   ---------------------------------------------------------------
   function Status_Of (The_Timer : in Kind) return Status is
   begin
      return The_Timer_Status (The_Timer);
   exception
      when others =>
         --[ Report the occurrence of an unknown fault.
         raise Unknown_Fault;
   end Status_Of;

   ---------------------------------------------------------------
   procedure Start (The_Timer        : in Kind;
                    With_The_Mode    : in Mode;
                    And_The_Interval : in Interval) is
      A_Choice : Choice;
   begin
      --[ Set the timer with the mode and interval.
      case A_Choice is -- the timer
         when Option1 => -- timer A
            --[ Set Timer A Mode.
            Statement;
         when others => -- timer B
            --[ Set Timer B Mode.
            Statement;
      end case;
   exception
      when others =>
         --[ Report the occurrence of an unknown fault.
         raise Unknown_Fault;
   end Start;

   ---------------------------------------------------------------
   procedure Wait_For (The_Timer : in Kind) is
   begin
      case The_Timer is
         when Timer_A =>
            --[ Get a Timer A interrupt service routine message.
            Statement;
         when Timer_B =>
            --[ Get a Timer B interrupt service routine message.
            Statement;
      end case;
      --[ Update the timer's Is_Active status.
      Statement;
   exception
      when others =>
         --[ Report the occurrence of an unknown fault.
         raise Unknown_Fault;
   end Wait_For;

end Api_Timer;
------------------------------------------------------------------
```

4.8.7 Api_Timer CSU

This package provides interval timing services to the client. Two hardware timers can be programmed:

- Timer A
- Timer B

The state transition diagram below shows the relationship between these operations, a timer's state, and error conditions:

```
                        Start
           +--------------------------------------+
  Stop     |                                      |
  +--+     |           +----+ Set or              |
  |  |     |           |    | Stop                |
  |  v     |           |    v                     v
+-----------+        +-----------+             +-----------+
|           | Set    |           | Start or    |           |Start
| Uninit-   |------->|   Idle    |------------>|  Active   |---+
| ialized   |        |           | Restart     |           |   |
| (e.g.     |        |           |             |           |   |
| powerup)  |        |           |             |           |   |
|           |        |           |             |           |   |
+-----------+        +-----------+             +-----------+   |
      |                    ^                         |         |
      | Start or           |              Stop       |         |
      | Restart            |           or Timeout    |         |
      v                    +-------------------------+         |
+-----------+                                                  |
|           |                                                  |
|  Error    |                                                  |
|           |<-------------------------------------------------+
|           |
|           |
+-----------+
```

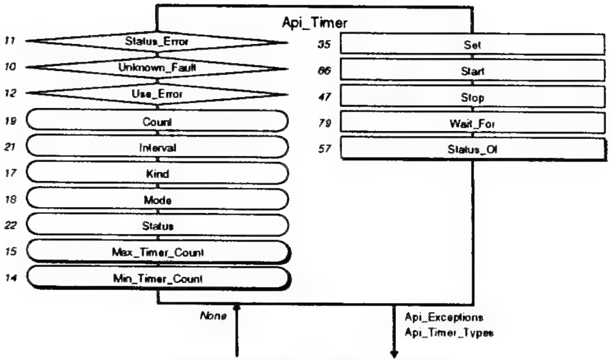


Figure 55. Api_Timer Diagram

4.8.7.1 Api_Timer Design Specification/Constraints

The requirements allocated to this CSU are identified in Section 7. The design requirements and constraints for this CSU are identified in the following specification.

The constraints and limitations for this unit are implicit in the specification that follows.

```
--------------------------------------------------------------------
with Api_Exceptions,
     Intel_Unsigned;

package Api_Timer is
   -----------------------------------------------------------------
   --| Initialization Exceptions: None.
   --| Version: 1.11  10/18/93
   -----------------------------------------------------------------
   Unknown_Fault  : exception renames Api_Exceptions.Unknown_Fault;
   Status_Error   : exception renames Api_Exceptions.Status_Error;
   Use_Error      : exception renames Api_Exceptions.Use_Error;

   Min_Timer_Count : constant := 100; -- microseconds
   Max_Timer_Count : constant := 16#FFFF_FFFF#; -- microseconds

   type Kind is (Timer_A, Timer_B);
   type Mode is (One_Shot, Continuous);
   type Count is new Intel_Unsigned.Integer32
      range 0 .. Max_Timer_Count;
   subtype Interval is Count range Min_Timer_Count .. Count'LAST;
   type Status (Is_Configured : Boolean := False) is
      record
         Is_Active : Boolean;
         Was_Reset : Boolean;
         case Is_Configured is
            when True =>
               Current_Mode     : Mode;
               Current_Interval : Interval;
            when False =>
               null;
         end case;
      end record;

   procedure Set (The_Timer        : in Kind;
                  To_The_Mode      : in Mode;
                  And_The_Interval : in Interval);
   -----------------------------------------------------------------
   --| Purpose:
   --|   Sets the mode and interval of a timer for the next call
   --|   to Restart.
   --| Exceptions:
   --|   Unknown_Fault - when the service traps an unexpected
   --|                   exception.
   -----------------------------------------------------------------

   procedure Stop (The_Timer  : in Kind;
                   Send_Pulse : in Boolean := False);
   -----------------------------------------------------------------
   --| Purpose:
   --|   Stops an active timer
   --| Exceptions:
   --|   Unknown_Fault - when the service traps an unexpected
   --|                   exception.
   -----------------------------------------------------------------

   function Status_Of (The_Timer : in Kind) return Status;
   -----------------------------------------------------------------
   --| Purpose:
   --|   Gets the current status of The_Timer.
   --| Exceptions:
   --|   Unknown_Fault - when the service traps an unexpected
   --|                   exception.
   -----------------------------------------------------------------

   procedure Start (The_Timer        : in Kind;
                    With_The_Mode    : in Mode;
                    And_The_Interval : in Interval);
   -----------------------------------------------------------------
   --| Purpose:
   --|   Starts The_Timer immediately after setting its mode and
   --|   interval.
   --| Exceptions:
   --|   Status_Error - when the timer is already active.
   --|   Unknown_Fault - when the service traps an unexpected
   --|                   exception.
   --------------------------------------------------------------

   procedure Wait_For (The_Timer : in Kind);
   --------------------------------------------------------------
   --| Purpose:
   --|   Suspends the caller until the specified timer reaches
   --|   terminal count.
   --| Exceptions:
   --|   Unknown_Fault - when the service traps an unexpected
   --|                   exception.
   --------------------------------------------------------------
end Api_Timer;
```

4.8.7.2 Api_Timer Design

```
Ada Unit Declarations:

   7 package body Api_Timer
       -------------------------------------------------- Visible Units
  12   procedure Set
  29   procedure Stop
  48   function Status_Of
  57   procedure Start
  74   procedure Wait_For
```

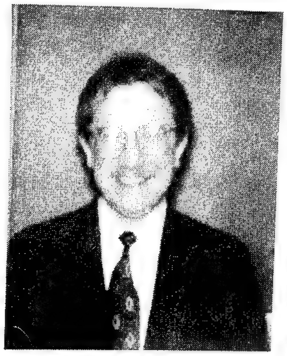
```
 1   --------------------------------------------------------------
 2   with Hdx_Interval_Timer;
 3
 4   with Pdl;
 5   use Pdl;
 6
 7   package body Api_Timer is
 8
 9      --------------------------------------------------------------
10      --| Visible units
11      --------------------------------------------------------------
12      procedure Set (The_Timer         : in Kind;
13                     To_The_Mode       : in Mode;
14                     And_The_Interval  : in Interval) is
15      begin
16         Save the configuration information for the timer.
17         case The_Timer is
18            when Timer_A =>
19               Write the interval to the Timer A module register
20            when Timer_B =>
21               Write the interval to the Timer B module register
22         end case;
23      exception
24         when others =>
25            Report the occurrence of an unknown fault.
26            raise Unknown_Fault;
27      end Set;
28      --------------------------------------------------------------
29      procedure Stop (The_Timer   : in Kind;
30                      Send_Pulse  : in Boolean := False) is
31      begin
32         if Timer_A is to be stopped then
33            Turn Timer A off.
34            if send pulse is True then
35               Place an event in the timer's ISR message queue.
36            end if;
37         else -- Stop Timer_B
38            if send pulse is True then
39               Place an event in the timer's ISR message queue.
```

```
40            end if;
41         end if;
42      exception
43         when others =>
44            Report the occurrence of an unknown fault.
45            raise Unknown_Fault;
46      end Stop;
47      --------------------------------------------------------------
48      function Status_Of (The_Timer : in Kind) return Status is
49      begin
50         return The_Timer_Status (The_Timer);
51      exception
52         when others =>
53            Report the occurrence of an unknown fault.
54            raise Unknown_Fault;
55      end Status_Of;
56      --------------------------------------------------------------
57      procedure Start (The_Timer         : in Kind;
58                       With_The_Mode     : in Mode;
59                       And_The_Interval  : in Interval) is
60      begin
61         Set the timer with the mode and interval.
62         case the timer is
63            when timer A =>
64               Set Timer A Mode.
65            when timer B =>
66               Set Timer B Mode.
67         end case;
68      exception
69         when others =>
70            Report the occurrence of an unknown fault.
71            raise Unknown_Fault;
72      end Start;
73      --------------------------------------------------------------
74      procedure Wait_For (The_Timer : in Kind) is
75      begin
76         case The_Timer is
77            when Timer_A =>
78               Get a Timer A interrupt service routine message.
79            when Timer_B =>
80               Get a Timer B interrupt service routine message.
81         end case;
82         Update the timer's Is_Active status.
83      exception
84         when others =>
85            Report the occurrence of an unknown fault.
86            raise Unknown_Fault;
87      end Wait_For;
88
89   end Api_Timer;
90   --------------------------------------------------------------
```

SUMMARY
Software Logistics - Knocking At The Army's Door

Software logistics is different than hardware logistics. The Army will have an estimated three hundred or more battlefield systems on the battlefield in the next ten years. There will be millions of software computer program media on the battlefield supporting these systems. Each system will have its requests for changes captured, consolidated, reviewed, incorporated into the next release, and then sent through the software logistics channels to the soldiers needing it. Each system will have to be managed to minimize or eliminate possible confusion and control the software's version currency for that system. Software changes are driven by a number of factors. Largely, the changes are made to meet changing requirements, stay useful, and better perform its mission. Changes serve the soldier on the battlefield. Without change, software would lose its greatest strength, its ability to address new situations quickly in emergencies and perform rapidly changing missions and thus meet requirements. Software change is driven by: new threats that must be countered; latent defects that must be corrected; doctrinal changes that must be incorporated; functional evolution that comes about from changing what is being done; interoperability requirements that necessitate battlefield system changes because an interfacing system has changed; technology changes that alter the system; hardware modifications that impact software; and safety mandates.

BIOGRAPHICAL SKETCH OF JOSEPH J. POTOCZNIAK



Mr. Joseph J. Potoczniak is Chief of the Replication, Distribution, Installation, and Training (RDIT) Group at the Communications-Electronics Command (CECOM), Research, Development, and Engineering Center (RDEC), Software Engineering Directorate (SED), Fort Monmouth, New Jersey 07703-5207, 908-532-0532, DSN 992-0532. He is a licensed Professional Engineer in the State of New Jersey and has over twenty-four years of experience with software and software systems. He holds a Bachelor of Science in Industrial Engineering (BSIE), a Master of Science in Management Science (MSMS), and a Master of Science in Electronics Engineering (MSEE). He has taken leave from his former position as Chief of the Operations and Maintenance Division at CECOM SED to organize and establish the RDIT concepts and objectives for CECOM.

SOFTWARE LOGISTICS - KNOCKING AT THE ARMY'S DOOR

Joseph J. Potoczniak, U.S. Army Communications-Electronics Command (CECOM); Research , Development, and Engineering Center (RDEC); Software Engineering Directorate (SED); Replication, Distribution, Installation, and Training (RDIT) Group; Fort Monmouth, New Jersey

In the past, electronic systems have been mostly hardware oriented with software used in a support role. Today, the roles are reversed. Software is the major part of most systems, with hardware providing the platform on which to operate the software. This is clearly evident in the U.S. Army Communications Electronics Command (CECOM) managed Battlefield Automated Systems (BASs).

What makes software different from hardware is that it will have to be changed rapidly, sometimes unpredictably, and often on a planned schedule. On the battlefield, software media will have to be replaced. The media may be broken, worn out, damaged, soiled, or affected by static electricity, but a percentage of the media in the field will have to be replaced.

Software changes are driven by a number of factors. Largely, the changes are made to meet changing requirements, stay useful, and better perform its mission. Changes serve the soldier on the battlefield. Without change, software would lose its greatest strength, its ability to address new situations quickly in emergencies and perform rapidly changing missions and thus meet requirements. Software change is driven by new threats that must be countered; latent defects that must be corrected; doctrinal changes that must be incorporated; functional evolution that comes about from changing what is being done; interoperability requirements that necessitate battlefield system changes because an interfacing system has changed; technology changes that alter the system; hardware modifications that impact software; and safety mandates.

These changes occur throughout the system life cycle. The changes to, or replacement of, software resident on CECOM-managed BASs frequently occur after the systems have been fielded. The actions required to provide those changes and replacements to fielded systems include replication, distribution, and installation of the software media; and the delta training of the system operators and other system-related personnel for these changes. These activities are referred to collectively as Replication, Distribution, Installation, and Training (RDIT).

By any other terms or acronyms, the process of RDIT is also the process of software logistics. It encompasses all of the preparation for getting the software to the battlefield, including the media, RDIT equipment, staffing, configuration control and monitoring, software resupply, installation, identification of software; and the use of fielding teams. Software logistics begins at the concept exploration phase of the BAS life cycle but is most active in Post Deployment Software Support (PDSS).

Software logistics is different from hardware logistics. We cannot have logistics business as usual for software. Of the Army's estimated 300 or more battlefield systems, in the next 10 years there are likely to be millions of software media on the battlefield. Each system will have its change requests recorded, consolidated, reviewed, incorporated into the next release, and then sent through the software logistics channels to the soldiers needing it. Each system will have to be managed to minimize or eliminate possible confusion and control the software's version currency for that system.

The procedures necessary to field new media or software changes are a large part of software logistics. Software logistics include post deployment software sustainment actions, iterative software matériel fieldings, and all the battlefield systems' concept, design, and early planning for software support over the life cycle. Yet, software logistics is not part of the Army's institutionalized programs, but is knocking at the door.

The reasons for the differences between software and hardware logistics are many. There can be no stockpiles of software for use on the battlefield because a deficiency in one copy of software would mean there is the same deficiency in the entire stockpile, hence invalidating any copies that would be kept in the stockpile. A failure in software is thus equivalent to a 100 percent failure of hardware. For the same reason, there can be no depot stockage of spare software.

The configuration control of software resupply is complicated more by volatility than is hardware. Software can have several versions in the field at the same time for the same battlefield system; this was true during Desert Storm. Software can be theater- and threat-dependent, and careful management must be exercised to accurately control specific version releases to specific units. In the worst case, some battlefield software will be unique to each of the individual systems. Software will be unique to each one of the individual systems. Software in general cannot be released without careful review of each release and resupply to the degree that only the software release authority (in practice, the software support facility is usually the acting surrogate) can authorize which version is to be issued to any particular unit for a particular battlefield system. Training is often required for software releases and must be conducted in coordination with each release to the field. While some releases are planned and scheduled, software releases are largely made in response to requirements during emergencies and are performed immediately.

The identification of software requires means different from hardware, which uses National Stock Numbers (NSNs), since software upgrades or changes can be made several times per year and must be visually identifiable. Firmware updates in the field never have the Printed Circuit Board's (PCB's) NSN changed, since the change of the NSN etched on the PCB is virtually impossible. Given that the NSN could be assigned, it is not visually identifiable by anyone needing to know what version is being examined.

In the next 10 years, there are likely to be millions of media on the battlefield. Each system will have to be managed in order to control the software version's currency. Adding to that, for any one system, different versions of software may be authorized for different theaters; additional versions may be allowed for particular exercises; and at any one time there will likely be at least one update of a system's software being started on any given day. Considering that a software update can take months to complete, the near future is likely to witness many different fielded systems being updated at any one time. Intensive management will be required due to this volatile characteristic of software. While peace-time may limit some updating activity, mobilization may dictate a groundswell of required changes to battlefield systems' software, as in Desert Storm.

If firmware fails or needs upgrading or replacement in one battlefield system, care must be taken to upgrade the system's printed circuit boards in all systems and in all depot and maintenance areas so that outdated and unauthorized versions of software are not unintentionally released to and/or used in the field.

Hardware is produced in quantity for spares and prepositioned; then, unlike software, the production processes are stopped and disbanded after the contract terminates. Software, not being stockpile driven but process driven, requires that the software sustainment of any battlefield system's software must constantly be available since a break in the sustainment means the dissolution of the software team and about two years of unaffordable time to reactivate a new team. The time requirement for having software upgraded on the battlefield can vary from hours to days to weeks, depending upon the mission that the system is performing (threat table software may require updates in the one-to-a-few-days time frame).

If a network is being upgraded with replacement software, care must be taken to coordinate the change so as not to degrade or obstruct mission capabilities. Another major difference is that software is changed much more frequently. It was reported that Patriot had three changes to the software during its use in Desert Storm. The Intelligence and Electronics Warfare (IEW) systems had over 30 changes during Desert Storm. Not changing software can directly translate to loss of life on the battlefield.

If battlefield systems are interoperating, a software change in one system may possibly require that a synchronized compatible change be made to the interoperating systems, even though there are no errors in those systems. In many cases, this coordinated release is done routinely and is mandatory. Hardware interoperability, on the other hand, is the exception, not the rule.

Readiness is used only when it is applied to hardware. Yet, software, which is in most of the systems on the battlefield, is not considered in the measure of readiness. A continuing priority requirement of the commander in the field is readiness. If software sustainability is not in place, or if the time to get software changes to the soldier is too lengthy to meet the needs of the battlefield, software readiness directly, and sometimes adversely, impacts the system's readiness. Readiness requirements dictate that, when needed, software changes and replacements are provided as rapidly and as timely as possible. It also means that the correct software version is provided if replacement of the media is required. However, the real measure is how quickly the software can be brought to the soldier in the field. (See Figure 1.)

This giant, called Software Logistics, that is knocking at the Army's door is visible in the form of many dedicated software trained individuals working independently of each other across the Army, much like fingers without a hand to connect them. It is visible in the costs that are higher than they would be if they were orchestrated through a common set of Army-wide standards and procedures. It is visible with expensive and scarce developmental software engineers carrying suitcases on planes to update firmware across the globe to do a technician's job. It is visible with many different battlefield systems having just as many different ways to solve the same software logistics problem and no standard approach in sight. It is hidden in the desks, personal computers, pocket reminders, and notebooks that are used to keep track of what software version is being shipped to whom and when; it is plainly visible by the growing concerns at many meetings about what to do with the problem of getting software from the program manager to the systems in the field.

The increasing number of fielded BASs and their worldwide distribution have resulted in inefficient ad hoc software logistics methods and have led to costly procedures and means for supporting the requirements of the soldier in the field. The nonexistence of a standard process for the replication and distribution of software and firmware to the field has resulted in one-of-a-kind, costly diversity of Computer Program Media (CPM) and procedures for each system and within functional areas such as air defense, fire support, etc.; reinventing the wheel; and proliferation of incompatible products such as replication equipment, CPM types, and software support environments.

These problems greatly increase the cost of support by creating a vendor dependency and the development of unique products instead of using nondevelopmental, off-the-shelf, or commercial items. These cost and performance inefficiencies point to the need for a centrally managed program with a defined process as the means to affect PDSS cost reduction and efficiencies. While the RDIT portion of the Software Logistics process necessitates central management, it does not require a central location for all work performed.

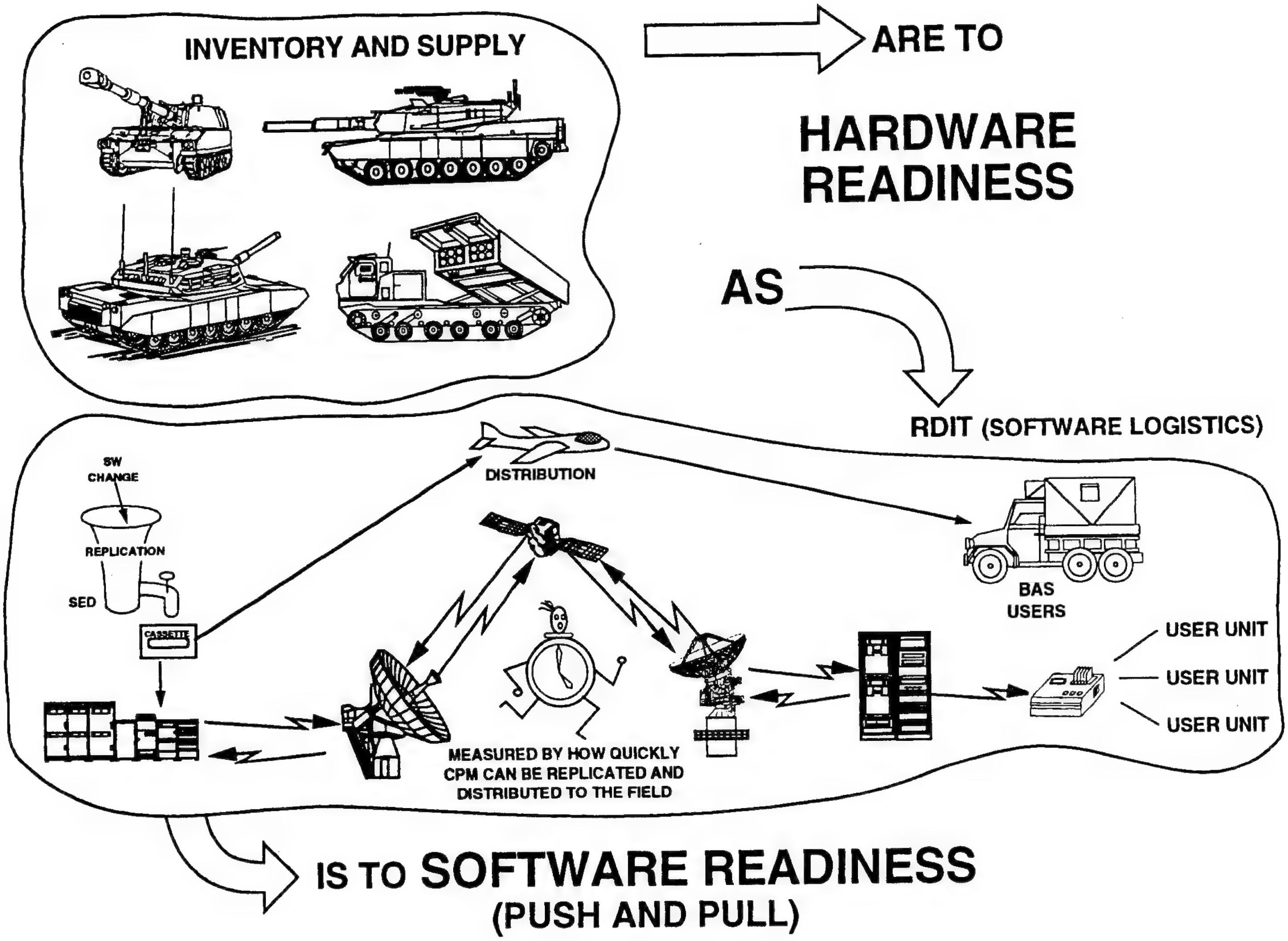
Through standardization of equipment, procedures, and management, cost savings can readily be obtained while actually improving the support to the soldier in the field. The current ad hoc system unique procedures, decisions, and practices need to be changed.

What order of savings can be accrued if the software RDIT practices were organized in the Army? Figure 2 shows that the cost of PDSS is 70 percent of the total Computer Resources Life Cycle Cost (CRLCC). Research has shown that RDIT is 25 percent of the PDSS cost or 18 percent of the CRLCC. However, these types of savings, in the millions or even billions of dollars can be achieved only through organizing the Army's Software Logistics program; a process of institutionalization involving the centralization of RDIT management.

Centralizing the RDIT process provides the advantage of bulk ordering of supplies; automated labeling and packaging; and better utilization of replication equipment, personnel, and facilities through support of multiple systems using the same media. The initial costs required to establish support for each new media type is more than offset by providing support to multiple systems. The more systems that use a common media and select CECOM Research, Development, and Engineering Center (RDEC) Software Engineering Directorate (SED) for RDIT support, the greater the cost savings to each system.

The earlier in the life cycle RDIT is planned, the more savings will accrue. For example, as shown in Figure 2, implementing the RDIT standard procedures and centralized management in the PDSS stage of the BAS life cycle saves 2 percent of the CRLCC. However, planning for RDIT functions in the concept exploration phase saves 16 percent of the CRLCC.

In the PDSS phase, savings are achieved by sharing the cost of maintaining and replacing standardized replication equipment with other Program Managers (PMs)/Program Executive Officers (PEOs), and sharing personnel and management costs. The cost of replication and distribution is reduced through automation that would not be affordable to individual systems, the use of a centralized tracking system to minimize the time and risk of fielding, the bulk purchase of media and



INVENTORY AND SUPPLY ARE TO HARDWARE READINESS AS RDIT (SOFTWARE LOGISTICS) IS TO SOFTWARE READINESS (PUSH AND PULL)

MEASURED BY HOW QUICKLY CPM CAN BE REPLICATED AND DISTRIBUTED TO THE FIELD

Figure 1. Readiness

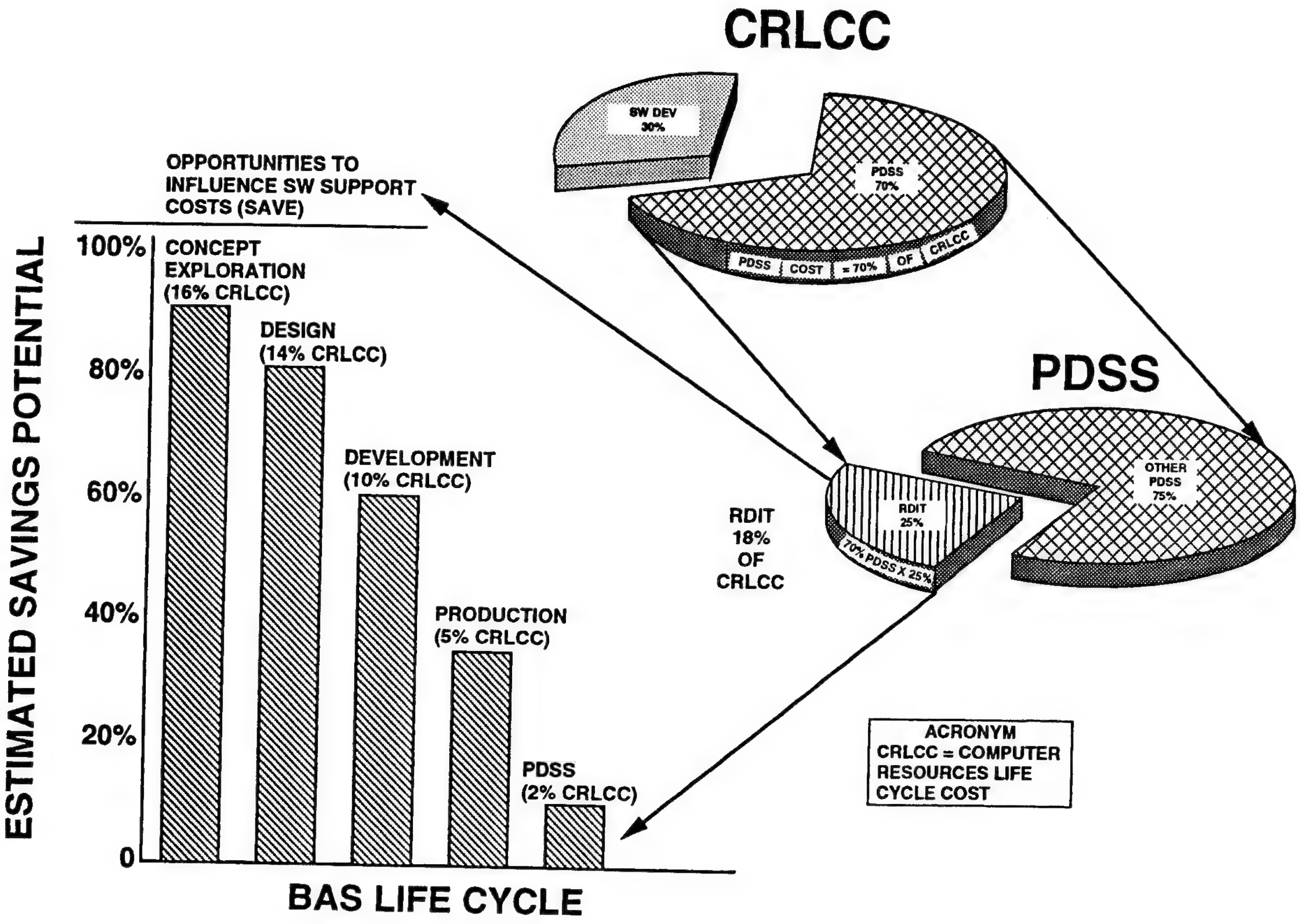


Figure 2. RDIT Cost Savings

packing materials, and standard procedures.

The savings in the production phase of the life cycle are a result of using well-trained software logistics planners who are familiar with the RDIT process and use methods that have proven to be efficient and effective. Also, the PM utilizes the existing RDIT facility, paying only a small part of the operating costs and thereby saving the time, cost, and risk of establishing an independently run facility.

During the development phase of the life cycle, the savings obtained are even greater. If a common media, common hardware, or common software is selected, initial production costs are greatly reduced. Since centralized RDIT would already have the physical facility, all the developing contractor would need to provide is the master media.

In the design phase, additional savings are realized from the increased Reliability, Availability, and Maintainability (RAM). If the design incorporates the use of common media and/or common hardware, as well as features that improve reliability and maintainability, savings will be even greater (up to 80 percent reduction in RDIT costs over systems that do not use centralized . RDIT support). This capability to influence standardization reduces the program cost and risk while easing the interoperability, logistics, and training burden placed on the Army.

The RDIT centralized management advises PMs on media and support environments that already exist within the Army inventory, ensuring that designs that require unique equipment are carefully evaluated as to the value added by the design versus the increased costs incurred to support unique requirements.

The greatest savings are realized when RDIT is considered during the concept phase of the life cycle. In addition to all of the savings of the previous phases, contracting costs and Request for Proposal (RFP) development costs and risks are reduced.

Once a battlefield system is produced and fielded, software begins a life cycle all its own. Software is developed, designed, changed, tested, released, and fielded repeatedly, long after the fielding of the battlefield system. Software fielding should be considered in the design of a battlefield system. The consequences of not doing so raise the likelihood that high costs and inadequate readiness may be unintentionally introduced into the system's support. The life cycle of a battlefield system must incorporate the deliberate plans for the support of software. Standard Statement-of-Work (SOW) clauses, software Integrated Logistics Support Plans (ILSPs), standard software support equipment (i.e., memory loader verifiers, PCB programmers, floppy disk replicators, high-volume tape copiers, Compact Disk Read-Only Memory [CD-ROM] replicators, etc.), standard software support procedures, Army regulations, and other means of reducing costs, increasing effectiveness, and improving readiness should play significant roles in how business is conducted, but it is not. The SED RDIT group at Fort Monmouth has, for over three years, been actively developing the concepts that form the foundation of software logistics.

The SED RDIT Group takes new version software to the soldier in the field by replicating it into the needed number of copies, assembling the corresponding document, packaging the kit, and ensuring that the kit gets to the right people in the right quantity at the right time. The software can be classified or unclassified on tapes, floppy disks, removable hard drives, CD-ROM, Ultraviolet Erasable Programmable Read-Only Memory (UVEPROM), Electrically Erasable Programmable Read-Only Memory (EEPROM), PCBs, or sealed in boxes. Regardless of the type of media on which the software resides, it is likely to change or need replacement. Software without RDIT would be like a factory without a shipping department. While the Army has a supply system and means of shipping hardware, at present the army is without a software shipping department. In addition, the SED RDIT Group is responsible for centrally managing SED's RDIT policies, processes, and procedures; planning for RDIT during software development; and, at the same time, providing a full range of RDIT services for fielded systems.

The SED, through its RDIT Group, is providing customers with a faster, resource-efficient, and cost-effective response to manage all software distribution, traceability, and resupply of the software commodity. The SED RDIT Group is an unequivocal leader in recognizing the importance of software logistics and providing cost-effective and mission-improving solutions.

Examples of SED RDIT Group Mission accomplishments are included in the following list:

a. Reduced costs and increased readiness by planning, identifying, testing, and evaluating state-of-the-art commercially available PCB programmers that resulted in the acquisition of an economical, automated, firmware replication capability for the SED.

b. Planned for and successfully tested a proof-of-concept for the implementation of the electronic transfer of software as a rapid, reliable, and economical method of distribution to the soldier in time of war or peace.

c. Developed extensive and detailed RDIT procedures easily adaptable to any specific system for use in planning for RDIT Army-wide.

d. Established new procedures and methods to bring Regency Net (RN) software and firmware to the units in Europe. This includes delineating actions and responsibilities for software/firmware changes and replacements. Procedures for resupply were also provided. Conducted three version distributions and numerous resupplies of RN media.

e. Established a new procedure for getting classified software to the soldier; prepared, tested, and is now using specialized procedures for handling and distributing classified software resupply and version changes. The process provides full and positive accountability, traceability, and delivery control from point of origin to final user destination.

f. Prepared and published procedures for conducting delta training, the only method by which the soldier can learn about the new software's capabilities.

g. Designed and implemented the computer-based Configuration Tracking System (CTS) to track to whom the software was given, when it was distributed, and other aspects of RDIT for fielding both new version and user-requested replacement computer program media. The CTS supports future RDIT planning, budget preparation, and management reports.

h. Prepared and published an unprecedented software ILSP model to facilitate the development and integration of 10 separate logistic support elements necessary to acquire, field, and support Army systems.

i. Prepared and published a much-needed Model Post Deployment Software Fielding Plan. This model provides information and guidance for all involved to ensure orderly and timely fielding plus continuing support of software after transition.

j. Continued research of flash memory card technology for a rapid reprogramming solution that is small, compact, reliable, and completely portable. The intent is to develop an enhanced and efficient method to reprogram BASs.

k. Developed and designed a software version identification system for CECOM software/firmware media to assist in the identification, management, and control of fielded software.

l. Prepared a Master Plan, Program Plan, Implementation Plan, and other documents that define and outline the overall responsibility of the SED in establishing and implementing software fielding procedures for CECOM. These plans elaborate upon how the management of the RDIT processes will be conducted.

m. Prepared a CECOM regulation that contributes to software readiness by assigning responsibilities for software fielding.

n. Reviewed and provided comments for over 60 regulations, policies, and procedures that help project managers save resources by including in their Computer Resource Life Cycle Management Plans (CRLCMPs) (or other appropriate planning documents) their approach to distributing new software/firmware versions to the field.

o. Developed methods for computing personnel resources required for fielding teams to install software/firmware and to conduct necessary delta training.

p. Investigated the feasibility of redefining how firmware is represented on engineering drawings. Currently, hundreds of drawings are used to define firmware at a high cost. A proposed procedure was developed and briefed to the Department of Defense (DoD) /Industry Drawing Practices Committee. As a result, the RDIT Group was made a member of a subcommittee on the Document Generation Process to further investigate the RDIT Group's proposal. Several of the SED RDIT Group recommendations have been adopted for inclusion in Military Standard 100F, entitled Engineering Drawing Practices. Preliminary computations demonstrate that the revised drawing practices will save millions of dollars for CECOM, and more for the Army and DOD.

q. The SED RDIT Group has continued to provide cost savings/cost avoidance to CECOM and SED Project Leaders. After setting the precedent by saving the PM, RN, $2.8 million on reprogramming devices, the RDIT Group followed up by saving $81,500 by replicating tape maps for the Maneuver Control System. This included $20,000 in bulk media purchase savings and $61,500 savings in labor costs.

r. During the past year, the RDIT Group has replicated 12,000 3 1/2-inch floppy disks for the AN/UGC-144 Communications Terminal. Given that the old replication method (utilizing three actual terminals) replicates 90 disks per hour (0.667 min. per diskette) and the RDIT method (utilizing a standard commercially available automated disk duplicator) replicates 400 disks per hour (0.15 min. per disk). The total savings in manpower costs and bulk media purchasing for diskettes was $9,600 or a 53 percent savings.

s. The savings realized for the AN/GSC-52, while not large in total dollars, was significant in manpower savings. The old method used to replicate, label, pack, and ship 650 diskettes took 46 man hours. The RDIT Group did the same job in 14 hours. The system's readiness is greatly enhanced by the RDIT Group.

As this is an emerging discipline knocking at the Army's door, more work needs to be done to address Software Logistics in the Army. Considering that software logistics consumes a large percentage of the life cycle costs of computer resources for battlefield systems, the efficiencies and efficacies that are being gained from the implementation of software logistics are notable and necessary to give the Army another advantage on the battlefield. The concepts, tools, and procedures that are forming the foundation of software logistics are rapidly evolving as more is learned.

Currently, many software logistics functions are performed in an ad-hoc manner, which results in each system having personalized (rather than organizational) service. These services are not institutionalized, not standardized, not documented, not addressed in regulations, and subject to the interpretations of individual system managers. The SED RDIT Group is opening the door to this giant called software logistics, and by doing so, is building a foundation for the orderly flow of software to the battlefield.

SOFTWARE PROCESS DEFINITION FOR AN ADA® PROGRAM WITH SIGNIFICANT SOFTWARE REUSE

Toni Shetler
TRW/Integrated Engineering Division
12900 Federal Systems Park Drive
Fairfax, VA 22033

Summary

This paper presents the software process definition proposed for a multiple contractor team to develop a large distributed instrumentation system residing on multiple platforms. Ada, according to MIL-STD-1815, was specified for all new software development and for reusable software with changes of 33% or more[1]. How the software development process was established is presented, along with the resulting process definition descriptions. The software development process is tailored DoD-STD-2167A and addresses Software Engineering Institute (SEI) capability maturity model (CMM) key processes[2]. It incorporates the following TRW Ada Process Model (APM) concepts: incremental development, design integration, demonstration-based technical reviews, and process improvement techniques[3]. It guides our integrated team approach to software development.

Introduction

A software development process provides a handbook description for how software will be created and is usually contained in a software development plan (SDP). While a well-defined process is important for any software development organization, it is *critical* for a project team comprising six corporations at separate locations, each having development responsibility for part of an integrated software product. This paper discusses the common software development process proposed for use by a TRW project software development team:

- motivation for the common process
- common process infrastructure
- process definition
- implementation considerations.

This paper describes the software development process baseline for the project. It is based on organizational process assets and experience. We expect to review, assess and improve this baseline throughout the project life. While we expect the process to be successful, we also expect it to change (improve) over the project life -- this paper provides the baseline for reporting progress.

Motivation

The target system software design is requirements driven, with the associated technical analyses and trade studies focused on how to achieve function and performance rapidly, at reduced risk and cost. Our software design team chose not to define computer software configuration item (CSCI) boundaries by contractor; instead, they proposed a software design that included substantial reuse, with a demonstrable initial software baseline (ISB) that included over sixty percent (60%) of the system's functionality and code. The software design is the result of:

- extensive use of team domain-specific knowledge, which crossed contractor boundaries. To take advantage of this knowledge requires sharing not only design, but also development responsibilities across CSCI boundaries.
- extensive reuse of existing components from the teammates, incorporated into CSCIs and computer software components (CSCs). This enables efficiencies not usually possible among multiple contractor teammates.
- adhering to Ada constraints on new and modified non-developmental software (NDS), while minimizing the *amount* of new code and modified NDS. While the test requirements remained constant, the over-all development effort is reduced by minimizing new and modified code.

The design effort focuses on three specific objectives:

- common intra-system software, where the same software provides common functionality across the system platforms
- software reuse, where commercial off-the-shelf (COTS) and other, non-Ada, NDS are integrated into the product
- domain knowledge and expertise, where work assignments align with teammate expertise and responsibility, and some component assignments cross CSCI boundaries.

Implementing the design requires a technical approach that assumes a *common* software process for the team, enabling it to function as a single, integrated software development unit. This approach is supported by other reuse experience, where software development efforts achieve substantive reuse by using an integrated software development team[4,5]. Strong motivation for the team to define and use a common process include:

- anticipated cost and schedule payoff from extensive reuse of system design and development components[6]
- risk mitigation associated with achieving function and performance objectives by reuse, incorporating proven components[7].

Infrastructure

The essential element for establishing a common process was the pre-existing, strong commitment of each teammate to software process. This commitment was based on an awareness and understanding of the Carnegie Mellon University (CMU) Software Engineering Institute (SEI) capability maturity model (CMM) and involvement in individual corporate SEI-based software process improvement efforts -- we had a common vocabulary to articulate concerns and exchange ideas[2,8]. Other key elements of the infrastructure that allow us to function as an integrated development team include:

- a common software development plan (SDP) to document processes and methods. The SDP was developed jointly by the team. As the controlling document for software management and technical development, it is consistent with TRW and its subcontractors' standard software development methods. It applies to newly developed software, the initial baseline of reused software (ISB), and to other NDS integrated into the system. It applies to all the system CSCIs, incorporates the common process definition, and addresses coding standards, metrics, configuration management procedures, quality control, and test standards.
- a common software engineering environment (SEE) that includes networked engineering tools and host and target environments. The SEE supports the following software engineering processes: 1) systems engineering, 2) software development and maintenance, 3) software test engineering, 4) software engineering management, and 5) software configuration management and quality assurance. The SEE repository was modeled logically -- using entity relationship diagrams (ERD) -- and is partitioned into the five areas, each owned by a software engineering process. The physical configuration of the SEE is a distributed network of facilities, inter-connected by commercial packet-switched circuits by way of gateways.
- project software engineering process group (SEPG) responsible for software process improvement, metrics, and training[2]. The project SEPG, consisting of team member representatives, is responsible for software process, including training and periodic process assessment. It was formed early in the project life cycle to ensure attention to the processes associated with building software, and to establish the framework for software process oversight -- definition, monitoring and improvement. The project SEPG was integral to defining the processes presented herein. It is also responsibility for common software training for all the software developers.

The project SEPG established a common software process, consistent with the prime contractor's (TRW) improvement process model. Figure 1 shows this model to improve product quality and contract performance. The elements of the closed-loop framework support our total quality management (TQM) objectives. Its application to software process improvement is indicated by the close parallels to the CMU/SEI software process CMM shown in the figure[3].

Software Process Definition

The common software process is based on incremental builds and features concurrent COTS/NDS analysis, evaluation and prototyping, integrated with traditional Ada analysis, design and development activities. The prime contractor's software process formed the basis for the team's process, with teammate processes incorporated as key features[9,10]. Integrating the team processes was simplified by substantial experience overlap from extensive DoD standards-based software development. The resulting common software process is a milestone-based, phased-development model that features incremental builds and tailored 2167A activities. This section includes:

- process flow diagrams that expand the tailored development life-cycle model for a detailed understanding of software work packages (Figure 2 illustrates the top-level process flow)
- process activity descriptions (PADs) that incorporate enhanced entry-task-exit (ETX) process descriptions. The PADs include key development elements: reviews and inspections, testing, software quality assurance (SQA), software engineering process group (SEPG), configuration management (CM), and events and milestones. Figures 3 through 8 illustrate the next level process flows from Figure 2 and includes software process, planning, and training activities.

Our baselined process incorporates concepts from the TRW Ada Process Model (APM): incremental development, design integration, demonstration-based technical reviews, and process improvement techniques[3].

Figure 2 shows the first-level expansion of the integrated software process. The development life cycle is at the top, with the tailored software process directly under it. Activities specific to our customized approach are shown relative to DOD-STD-2167A milestones[11]. The activities (and their relationships) that lead to milestones are expanded in the center of the figure. The value of Ada properties that facilitate building infrastructure and encapsulation is indicated by COTS/NDS analysis, evaluation, prototyping, modification and validation activities in the process. The unique (and important) feature of this approach is using demonstrations and prototyping to grow products. Each activity shown is further detailed in Figures 3 through 8 as entry/task/exit (ETX) descriptions, expanded activity flows, and process, planning, and training activities.

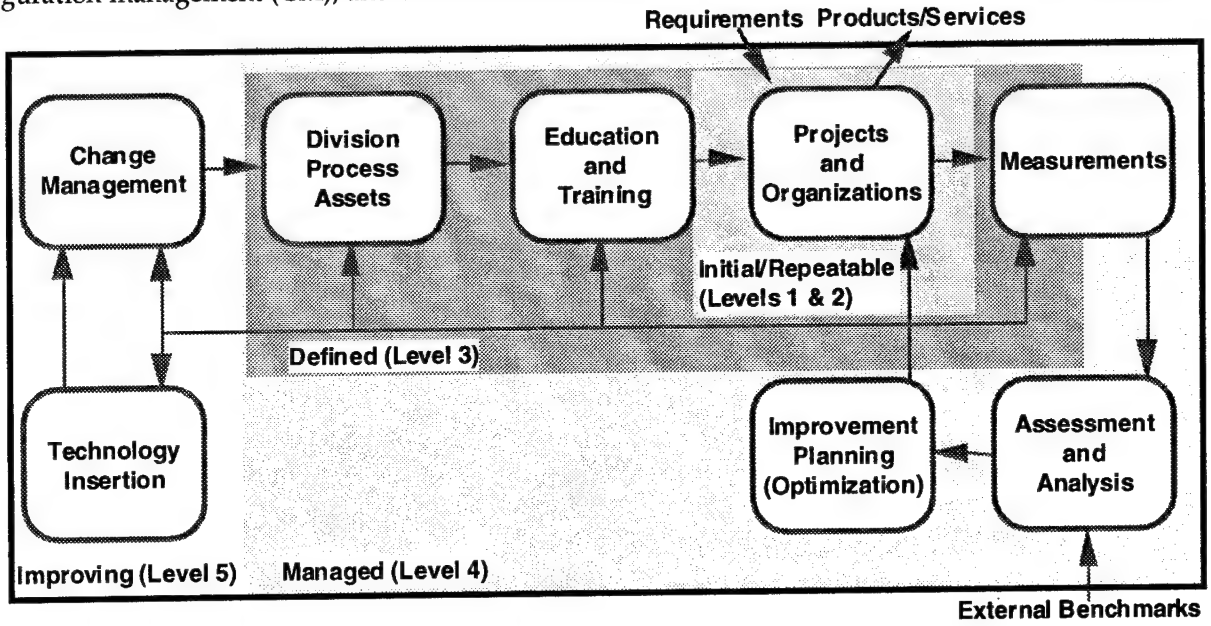


Figure 1. TRW Closed-Loop Process Improvement Framework

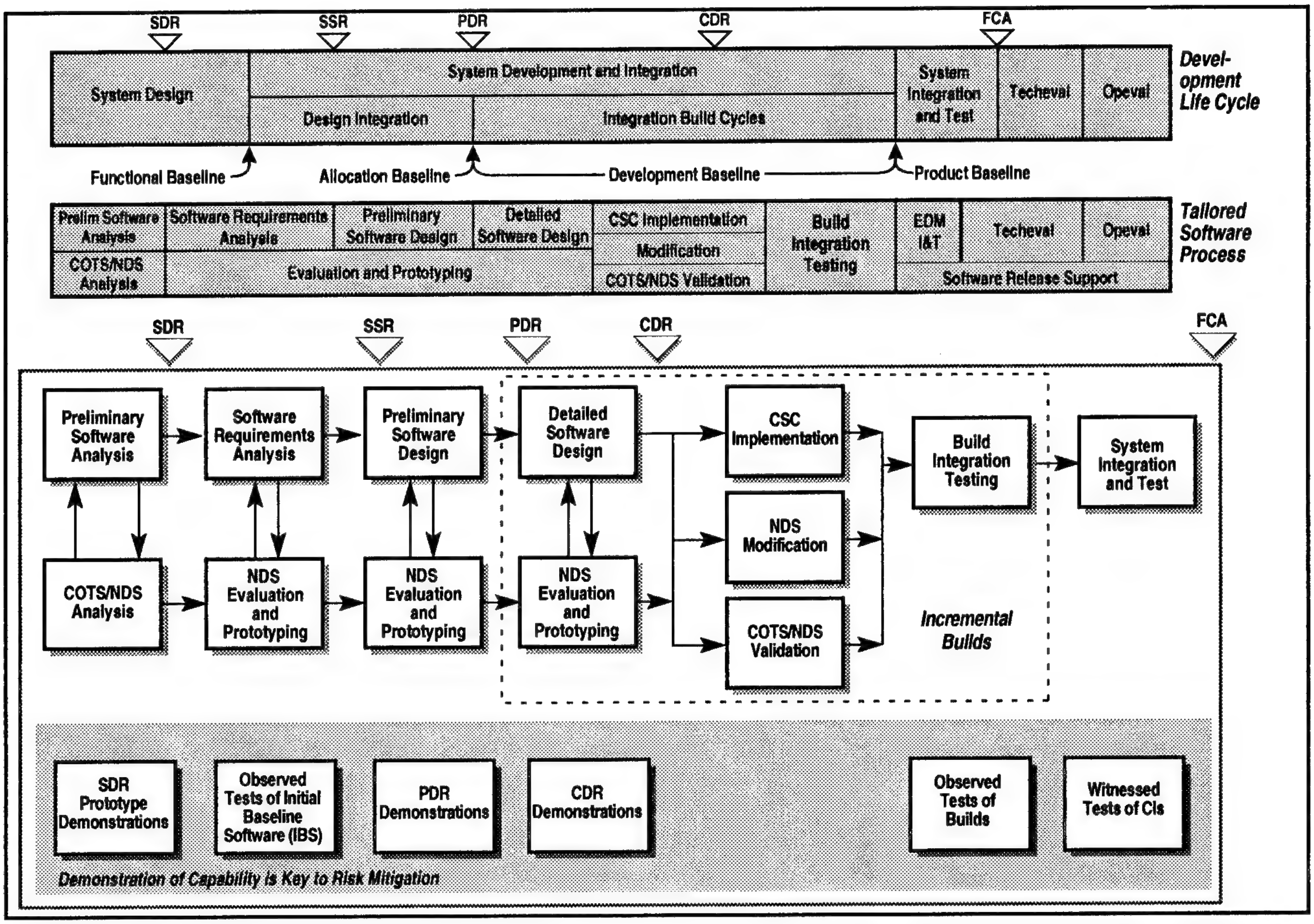
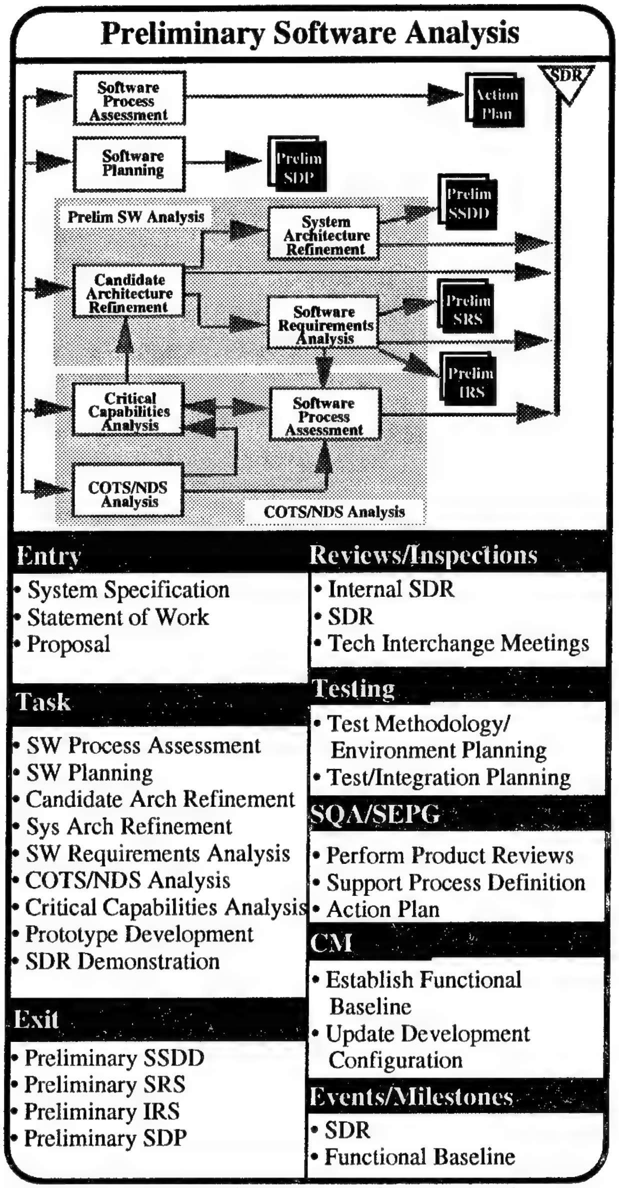

SDR
SSR
PDR
CDR
FCA
System Design
System Development and Integration
Design Integration
Integration Build Cycles
System Integration and Test
Techeval
Opeval
Devel-opment Life Cycle
Functional Baseline
Allocation Baseline
Development Baseline
Product Baseline
Prelim Software Analysis
Software Requirements Analysis
Preliminary Software Design
Detailed Software Design
CSC Implementation
Modification
COTS/NDS Validation
Build Integration Testing
EDM I&T
Techeval
Opeval
COTS/NDS Analysis
Evaluation and Prototyping
Software Release Support
Tailored Software Process
SDR
SSR
PDR
CDR
FCA
Preliminary Software Analysis
Software Requirements Analysis
Preliminary Software Design
Detailed Software Design
CSC Implementation
Build Integration Testing
System Integration and Test
NDS Modification
COTS/NDS Analysis
NDS Evaluation and Prototyping
NDS Evaluation and Prototyping
NDS Evaluation and Prototyping
COTS/NDS Validation
Incremental Builds
SDR Prototype Demonstrations
Observed Tests of Initial Baseline Software (IBS)
PDR Demonstrations
CDR Demonstrations
Observed Tests of Builds
Witnessed Tests of CIs
Demonstration of Capability is Key to Risk Mitigation

Figure 2. Top-Level Process Flow for the Software Process

The process employs early prototyping and evaluation. To accommodate this process we recommended tailoring DOD-STD-2167A so that: 1) COTS, NDS, or government furnished software (GFS) is integrated into the demonstration and design prior to critical design review (CDR); 2) CSC qualification occurs during build thread testing; 3) aggregate thread testing is substituted for formal CSCI testing at the build level, and 4) software documentation requirements are modified by Data Item Description (DID) tailoring[11,12]. The methodology to support these changes uses demonstration-driven reviews and prototyping to evaluate/integrate COTS, NDS, GFS, and new code into the development baseline.

The overall rationale is risk reduction by using proven and demonstrable products, with early insight and visibility into development progress[7]. Risk reduction results from having a solid baseline to prove functionality and to remove the uncertainty of *future-ware*. Early visibility by demonstrating development progress provides a clear understanding of the requirements of the operational system, what progress has being made, and how the requirements are manifested in operational software. Additional tailoring rationale lies in the approach to software testing. By adopting a distributed processing system and a layered approach to CSCI definition, formal software requirements testing is realized by exercising and testing executable threads during builds. This avoids the *big bang* effect that accompanies traditional formal CSCI testing, which does not apply when all CSCIs must be present/operational to prove the functional and performance requirements of any one CSCI.


Preliminary Software Analysis
Software Process Assessment
Action Plan
SDR
Software Planning
Prelim SDP
Prelim SSDD
Prelim SW Analysis
System Architecture Refinement
Candidate Architecture Refinement
Software Requirements Analysis
Prelim SRS
Prelim IRS
Critical Capabilities Analysis
Software Process Assessment
COTS/NDS Analysis
COTS/NDS Analysis

Entry
- System Specification
- Statement of Work
- Proposal

Task
- SW Process Assessment
- SW Planning
- Candidate Arch Refinement
- Sys Arch Refinement
- SW Requirements Analysis
- COTS/NDS Analysis
- Critical Capabilities Analysis
- Prototype Development
- SDR Demonstration

Exit
- Preliminary SSDD
- Preliminary SRS
- Preliminary IRS
- Preliminary SDP

Reviews/Inspections
- Internal SDR
- SDR
- Tech Interchange Meetings

Testing
- Test Methodology/ Environment Planning
- Test/Integration Planning

SQA/SEPG
- Perform Product Reviews
- Support Process Definition
- Action Plan

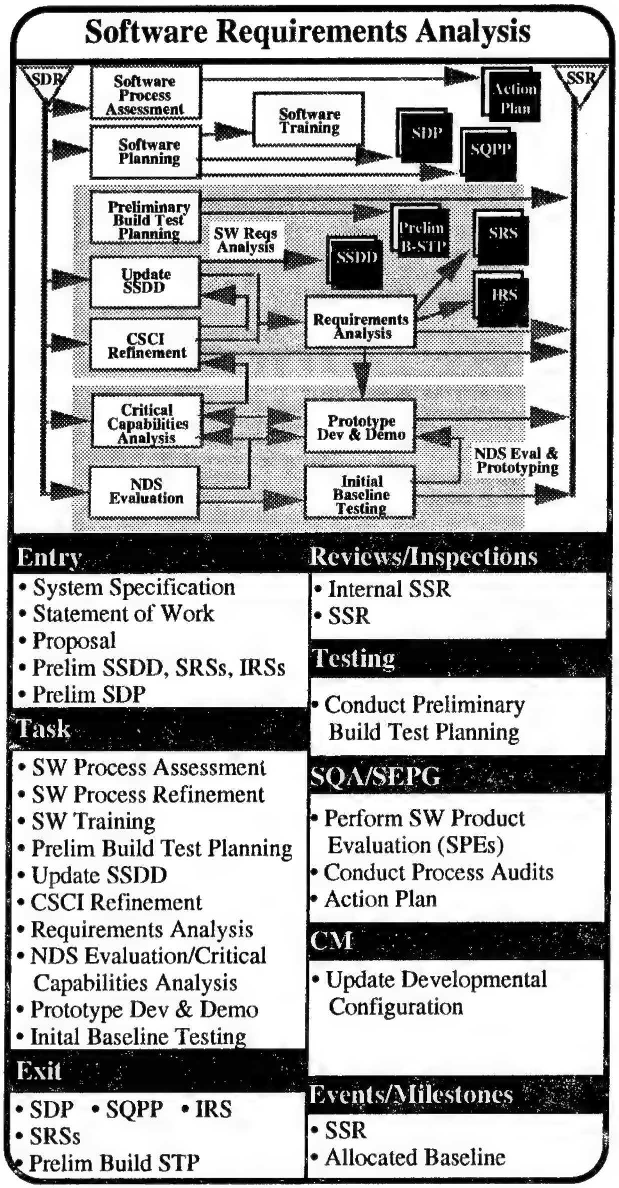
CM
- Establish Functional Baseline
- Update Development Configuration

Events/Milestones
- SDR
- Functional Baseline

Figure 3. Preliminary Software Analysis

Preliminary Software Analysis

This phase includes: the preliminary software analysis for the total software system and new development, and the COTS/NDS analysis. These activities include refinement of the candidate architecture and system architecture, software requirements analysis, critical capabilities analysis, COTS/NDS analysis, prototype development and SDR demonstration. The result is preliminary documentation (SSDD, SRSs and IRSs) and prototype demonstrations at SDR. Demonstrations mitigate risk associated with relying on paper descriptions of the product during the early stages of development. An integral part of this activity is development of the functional baseline.



Entry
- System Specification
- Statement of Work
- Proposal
- Prelim SSDD, SRSs, IRSs
- Prelim SDP

Task
- SW Process Assessment
- SW Process Refinement
- SW Training
- Prelim Build Test Planning
- Update SSDD
- CSCI Refinement
- Requirements Analysis
- NDS Evaluation/Critical Capabilities Analysis
- Prototype Dev & Demo
- Inital Baseline Testing

Exit
- SDP • SQPP • IRS
- SRSs
- Prelim Build STP

Reviews/Inspections
- Internal SSR
- SSR

Testing
- Conduct Preliminary Build Test Planning

SQA/SEPG
- Perform SW Product Evaluation (SPEs)
- Conduct Process Audits
- Action Plan

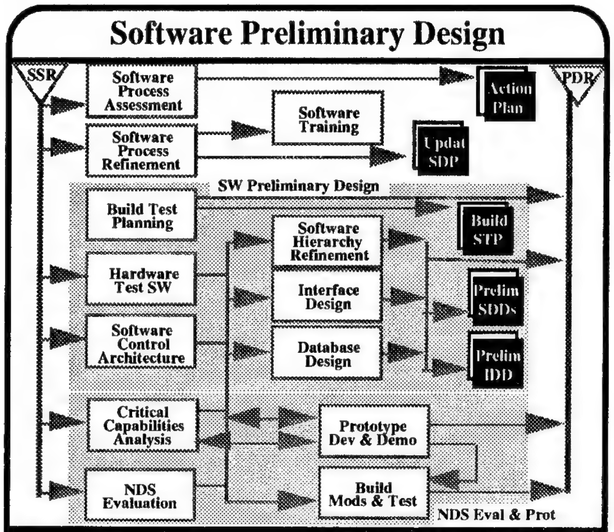
CM
- Update Developmental Configuration

Events/Milestones
- SSR
- Allocated Baseline

Figure 4. Software Requirements Analysis

Software Requirements Analysis

This phase includes: software requirements analysis and NDS evaluation and prototyping. These activities include refinement of the CSCIs, updating the preliminary documents, requirements analysis, critical capabilities analysis, NDS evaluation, prototype development and demonstrations, and initial baseline testing. The result is a preliminary build software test plan, final documentation (SSDD, SRSs and IRSs), and prototype demonstrations at SSR. An integral part of this activity is development of the allocated baseline.



Entry

- NDS • SDP • SRSs • IRS
- Prelim Build STP
- Schedule • Prototype Results

Task

- SW Process Assessment
- SW Process Refinement
- SW Training
- Build Test Planning
- HW Test Software
- SW Control Architecture
- SW Hierarchy Refinement
- Interface Design
- Database Design
- Critical Capabilities Anal/ Prototype Dev & PDR Demo
- NDS Evaluation
- Modify & Test Build

Exit

- SPEs • SDP •STP
- Prelim SDDs & IDD
- Build Threads Defined

Reviews/Inspections

- Evaluate Prelim Design/ CSCs, Req Compliance, AIs, Data Flow (Processes/ Tasks & Global Data Defs
- Internal PDR • PDR

Testing

- CSC Test Requirements
- Build Test Planning

SQA/SEPG

- Perform SPEs
- Conduct Process Audits
- Action Plan

CM

- Establish Allocated Baseline
- Update Development Config
- Establish SDFs & IDFs (CSCs and Build)

Events/Milestones

- PDR

Figure 5. Preliminary Software Design

Preliminary Software Design

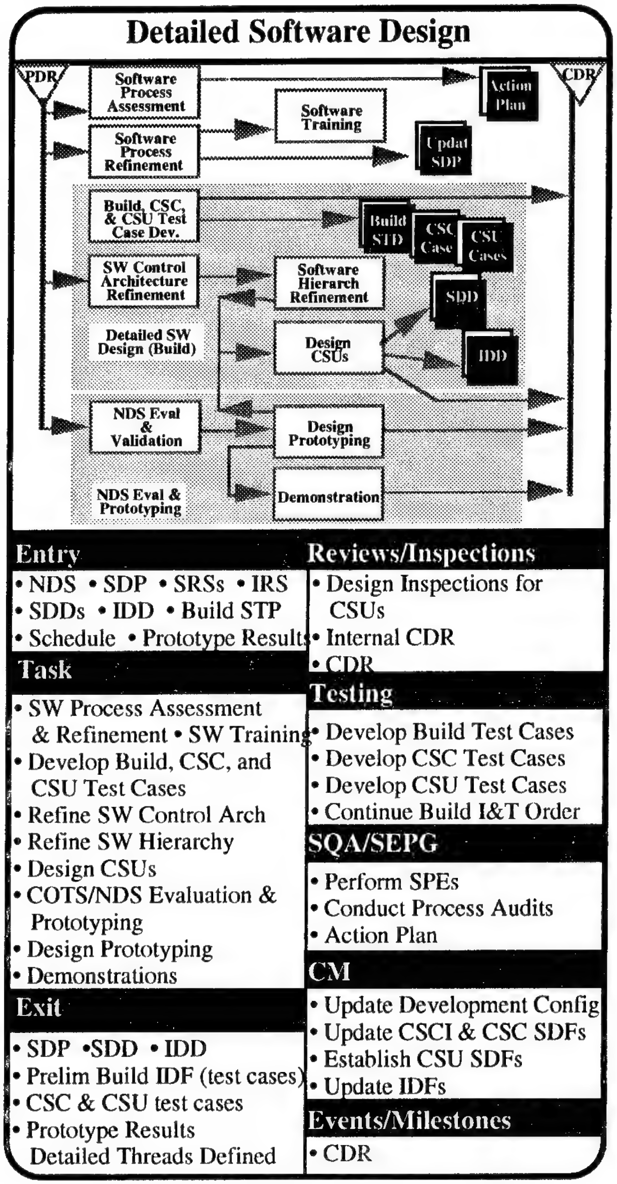
This phase includes: design process for new development and the continued evaluation and prototyping of NDS. These activities include critical capabilities analysis, NDS evaluation, prototype development, development, and test necessary to transform the ISB into the Build baseline, and demonstrations of capabilities at reviews. The result is a logical structure for the software, which defines (or refines) the algorithms and data needed to implement capabilities specified by the software requirements analysis activity. An integral part of this structure is the NDS baseline for the next build activity. The logical structure reported at preliminary design is computer software components (CSCs), although a more mature product will be available, including the CSU definitions.

Incremental Build Cycle

This phase provides a structured approach to phased implementation of detailed design, NDS modification, COTS validation, code development, build integration, and testing. Each build is composed of a set of related software functions, or *threads*, that satisfy specific software requirements. Build capabilities are defined by execution threads and corresponding requirements, which are satisfied by the build. Builds accomplish phased implementation of software CSCIs in a way that reduces risk by testing and demonstrating the maturing product. Builds are designed and implemented to the target system or a subset of the target system, with each build using the previous build as its foundation for construction. Each successive build adds functional capabilities that respond to increasingly larger subsets of the software requirements. The final build provides the functional capabilities responsive to all software requirements. Because the Test and Evaluation organization has build testing responsibilities, the final testing effort becomes a demonstration of qualification testing.

The incremental build development process activities are shown in Figures 6, 7 and 8, with activity process flows and expanded ETX descriptions. The incremental build-cycle activities correspond to the DOD-STD-2167A Detailed Design, Code and CSU Testing, and CSC Integration and Test activities[11]. The 2167A Detailed Design activity is addressed by the Detailed Software Design PAD. The 2167A Code and CSU Testing activity is addressed by the Software Build Implementation PAD. The 2167A CSC Integration and Test activity is addressed by the Integration Build Testing PAD. As was noted, the 2167A requirement for CSCI formal qualification testing (FQT) has been replaced by Integration Build Testing, where the Test and

Evaluation organization conducts Build Test Readiness Reviews (TRRs) prior to the conduct of Build Testing.



Entry

- NDS • SDP • SRSs • IRS
- SDDs • IDD • Build STP
- Schedule • Prototype Result

Task

- SW Process Assessment & Refinement • SW Trainin
- Develop Build, CSC, and CSU Test Cases
- Refine SW Control Arch
- Refine SW Hierarchy
- Design CSUs
- COTS/NDS Evaluation & Prototyping
- Design Prototyping
- Demonstrations

Exit

- SDP •SDD •IDD
- Prelim Build IDF (test cases)
- CSC & CSU test cases
- Prototype Results Detailed Threads Defined

Reviews/Inspections

- Design Inspections for CSUs
- Internal CDR
- CDR

Testing

- Develop Build Test Cases
- Develop CSC Test Cases
- Develop CSU Test Cases
- Continue Build I&T Order

SQA/SEPG

- Perform SPEs
- Conduct Process Audits
- Action Plan

CM

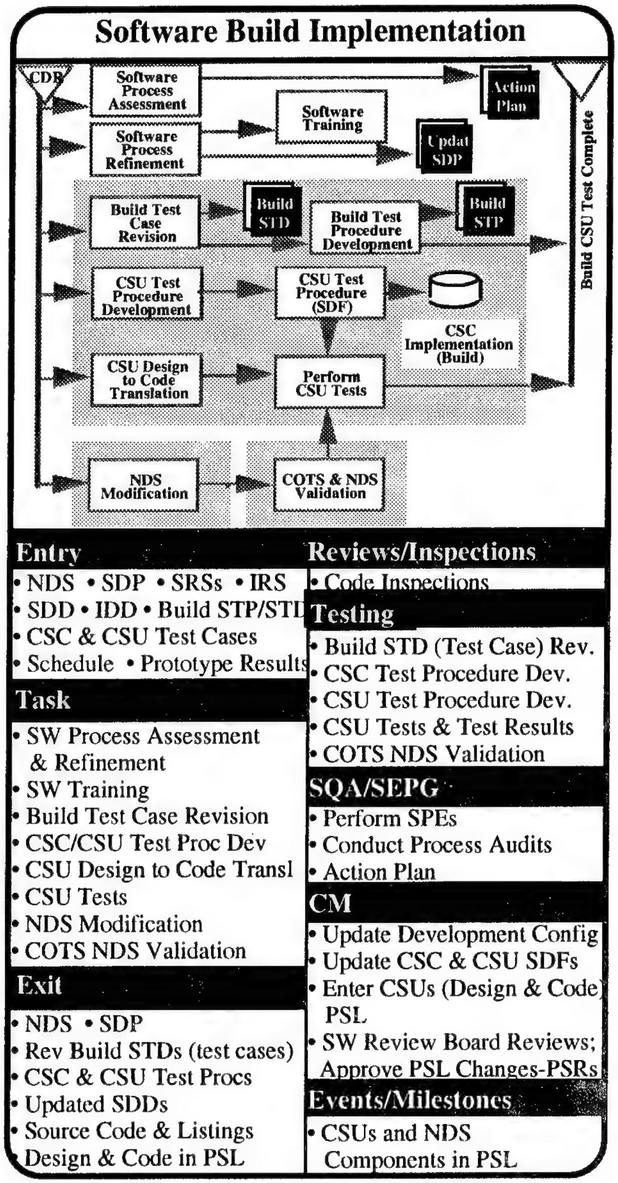
- Update Development Config
- Update CSCI & CSC SDFs
- Establish CSU SDFs
- Update IDFs

Events/Milestones

- CDR

Figure 6. Detailed Software Design

Detailed Software Design accepts the preliminary software design and associated design activity data, refining it to a complete definition of each CSU. The complete modular, lower-level design of the CSCIs is thus prescribed by the CSU design, which includes decomposing CSCs into CSUs, identifying critical CSUs, developing and refining database structures and controls, updating timing and sizing estimates, developing prototypes for high-risk software, and performing degraded performance and recovery analysis. NDS code and databases are treated as objects with a simple uniform interface to a main controlling CSC or CSU, which maximizes reusable components through connection ports that encapsulate classes of dissimilar NDS with the system.


Software Build Implementation
CDR
Software Process Assessment
Action Plan
Software Training
Software Process Refinement
Updat SDP
Build CSU Test Complete
Build Test Case Revision
Build STD
Build Test Procedure Development
Build STP
CSU Test Procedure Development
CSU Test Procedure (SDF)
CSC Implementation (Build)
CSU Design to Code Translation
Perform CSU Tests
NDS Modification
COTS & NDS Validation

Entry

- NDS • SDP • SRSs • IRS
- SDD • IDD • Build STP/STI
- CSC & CSU Test Cases
- Schedule • Prototype Results

Task

- SW Process Assessment & Refinement
- SW Training
- Build Test Case Revision
- CSC/CSU Test Proc Dev
- CSU Design to Code Transl
- CSU Tests
- NDS Modification
- COTS NDS Validation

Exit

- NDS • SDP
- Rev Build STDs (test cases)
- CSC & CSU Test Procs
- Updated SDDs
- Source Code & Listings
- Design & Code in PSL

Reviews/Inspections

- Code Inspections

Testing

- Build STD (Test Case) Rev.
- CSC Test Procedure Dev.
- CSU Test Procedure Dev.
- CSU Tests & Test Results
- COTS NDS Validation

SQA/SEPG

- Perform SPEs
- Conduct Process Audits
- Action Plan

CM

- Update Development Config
- Update CSC & CSU SDFs
- Enter CSUs (Design & Code) PSL
- SW Review Board Reviews; Approve PSL Changes-PSRs

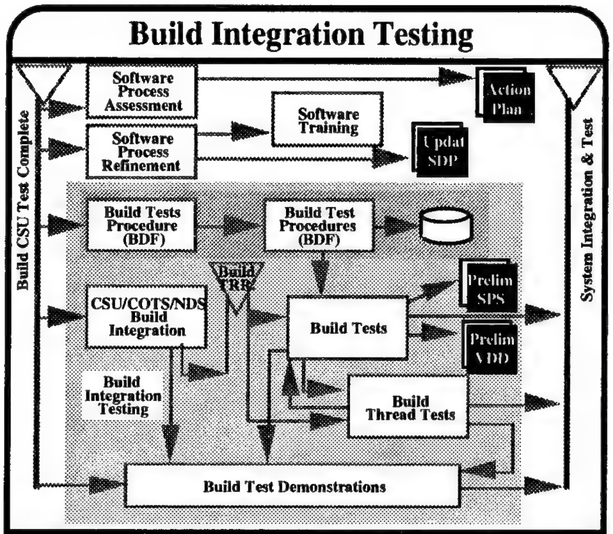
Events/Milestones

- CSUs and NDS Components in PSL

Figure 7. Build Implementation

Software Build Implementation begins at completion of a Critical Design Review (CDR). Some new developed software will be implemented in Ada, with exceptions for database query interfaces, graphical user interfaces, firmware, and modification to NDS components less that 33 percent of the code. Test procedures for CSUs and

NDS components, as well as build test cases and build test procedures, are developed in parallel with the software.


Build Integration Testing
Build CSU Test Complete
Software Process Assessment
Software Process Refinement
Software Training
Action Plan
Updat SDP
Build Tests Procedure (BDF)
Build Test Procedures (BDF)
Build TRR
CSU/COTS/NDS Build Integration
Build Tests
Prelim SPS
Prelim VDD
Build Integration Testing
Build Thread Tests
Build Test Demonstrations
System Integration & Test

Entry

- SDP • Program Support Lib
- Build STDs (Test Cases)
- CSC Test Procedures
- CSU and NDS Test Results

Task

- SW Process Assessment & Refinement
- SW Training
- Build Test Procedure Dev
- Integrate CSUs and NDS Components into Builds
- Conduct Build Test
- Gen Prelim SPSs and VDDs
- Test Build Threads
- Product Demonstrations

Exit

- SDP • Build STDs
- Prelim SPSs & VDDs
- Build Test Results
- Build SW Test Reports
- Updated SW Products
- SW Product Baseline

Reviews/Inspections

- Internal Build TRR
- Build TRRs (Builds 1

Testing

- Build Test Procedure Dev
- Conduct CSC Tests
- Conduct Build Tests
- Conduct Build Thread Tests

SQA/SEPG

- Perform SPEs
- Conduct Process Audits
- Action Plan (Build 1 to n-1)
- Lessons Learned (Build n)

CM

- Update Development Config
- Update PSL
- Estab SW Build Baseline
- Estab SW Product Baselines (Build 1 through n)

Events/Milestones

- Build TRRs • Thread Tests
- SW Product Baseline

Figure 8. Build Integration Testing

Build Integration Testing corresponds to the DOD-STD-2167A CSCI Integration and Test activity. Upon completing testing, coded units and NDS components are logically assembled to demonstrate the defined build functional capabilities of the software. Code development is synchronized with the build process to ensure delivery for integration and demonstration of the desired build functional capabilities. Over time, additional units of code and NDS components are added to provide increased software capabilities.

The build implementation methodology accommodates a software development cycle that is visible, controllable, and demonstrable. It facilitates regression testing and provides an effective method for controlling code as it goes through the test process. Build thread tests are conducted to demonstrate specific required capabilities across both hardware and software. Product demonstrations are performed to demonstrate system functionality.

Successful completion of the final build establishes the preliminary software product baseline and completes the incremental build cycles. Furthermore, the aggregate of build thread testing serves as the logical substitute for CSCI testing when formally performed and documented. At this time the preliminary Software Product Specifications (SPSs) and Version Description Documents (VDDs) are generated to document the product baseline.

The build implementation methodology is expected to result in highly stable, mature software configuration for formal testing, and a TRR is conducted with the customer prior to the start of formal testing.

Implementation Considerations

Inclusion and consensus are the principal implementation techniques used to define the common software development process and establish the mechanism for continuous process improvement. Inclusion addresses how we brought together representatives from separate corporate cultures to form an integrated software process definition team. Consensus was used to foster participant commitment to, and ownership of, the software development process; it involved:

- formulating and gaining acceptance from our geographically dispersed integrated work team to use a common software process for developing software
- accommodating varied corporate cultures with their separate approaches to key technical issues such as methodology, training, and tool selection for the common software engineering environment (SEE)

- establishing a single, integrated project SEPG, with participation from all of the teammates involved in developing, integrating and testing software
- designing a software process based on extensive reuse that addresses the Ada requirement for new development and extensive modification of NDS.

Inclusion involved three elements -- selection, training and team building. Each contractor selected its participant(s) by designating either the person who would lead their portion of the software development effort, or a person who had the professional respect of their development lead. The participants were familiar with CMU/SEI and other related process materials. Training involved participation in classroom, one-on-one, and workshop sessions, where the objectives were focused on developing the process definition materials and achieving consensus. Temporary collocation during process definition and joint meetings to review materials contributed substantially to achieving consensus on the software process baseline. Team building has been a natural by-product of working together, technical interchange, and the commitment to consensus.

Our vehicle for long term consensus is the project SEPG, whose membership includes representatives from system engineering, integration and test, configuration management, and quality assurance, as well as the leadership from each teammate's software development unit[13]. The project SEPG provides the continuity and project visibility for software process, and has definition and review responsibilities for:

- software process, methods and metrics
- software engineering environment (SEE)
- project SDP and software quality program plan (SQPP)
- software training needs and plans
- software development procedures and standards.

It additionally has responsibilities for:

- project-specific awareness and process training
- reporting on and sponsoring process improvement activities
- promoting process technology improvements
- developing training materials
- performing periodic process assessments and recommending improvements.

Conclusions

Because the project is at an early stage, conclusions about specific processes is premature. However, the work to date is promising, and publication of the software process baseline provides a vehicle against which assessments and improvement efforts can be measured.

Acknowledgments

While errors in preparation and presentation of this paper rest with the author, several people deserve acknowledgment for their contribution. James W. Neeley, Jr., Lockheed Sanders Corporation, contributed substantially to the common software process described herein. Susan D. Markel, TRW/SIG-East SEPG, provided significant commentary. And John M. Gormally, the TRW project program manager, provided valuable encouragement and guidance.

References

1 MIL-STD-1815A, *Ada Programming Language,* dated 22 January 1983.

2 Paulk, Mark C., Bill Curtis, Mary Beth Chrisses, and Charles V. Weber, *Capability Maturity Model for Software, Version 1.1,* Software Engineering Institute CMU/SEI-93-TR-24, February 1993.

3 Royce, W.E., "TRW's Ada Process Model For Incremental Development of Large Software Systems", *Proceedings of the 12th International Conference on Software Engineering,* Nice France, March 1990.

4 Waligora, Sharon, and James Langston, "Maximizing Reuse: Applying Common Sense and Discipline", pp. 161-180, *Seventeenth Annual Software Engineering Workshop*, Software Engineering Laboratory, Goddard Space Flight Center, Greenbelt, MD, December 1992.

5 Wessale, W, D. Reifer, and D. Weller, "Large Project Experiences with object-Oriented Methods and Reuse", pp. 204-225, *Seventeenth Annual Software Engineering Workshop*, Software Engineering Laboratory, Goddard Space Flight Center, Greenbelt, MD, December 1992.

6 Bond, Robert T., "Large-Scale Software Reuse in Naval Systems: A Case Study (Bofors Electronics' FS 2000 System), *AFCEA West*, 23-25 April, 1991, San Diego, CA.

7 Boehm, Barry W., "Software Risk Management: Principles and Practices", *NSIA Joint Risk Management Conference*, Los Angeles, CA 30 September 1987.

8 Humphry, Watts, *Managing the Software Process*, Addison-Wesley Publishing Co., 1990.

9 Software Engineering Process Group (SEPG), *TRW/SIG East Software Development Handbook*, 1993.

10 TRW/SD SEPG, *Systems Division Software Engineering Processes*, 2 June 1992.

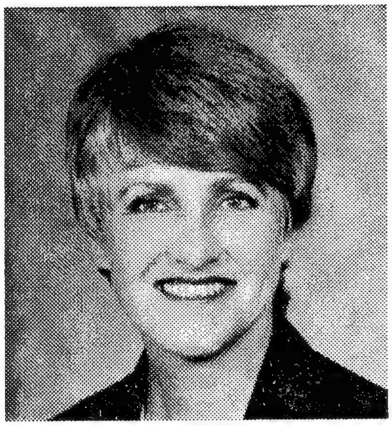
11 DOD-STD-2167A, *Defense System Software Development*, dated 29 February 1988.

12 DOD-STD-2168, *Defense System Software Quality Program*, dated 29 April 1988.

13 Putnam, Lawrence H., Arlyn D. Schumaker and Paul E. Hughes, *Economic Analysis of Re-Use and Software Engineering Process*, Volume I and II, TR-9265/11-2, Final Draft Report prepared for Standard Systems Center, Air Force Communications Command, Maxwell air Force Base, Alabama 36114, 10 February 1993.

14 Boehm, Barry W., *Software Engineering Economics*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

15 Air Force Systems Command, *Air Force Systems Command Software Management Indicators*, AFSC Pamphlet 800-43, 31 January 1986.



Toni Shetler is a senior engineer for TRW's Military Business Segment, and is deputy to the program manager for training instrumentation system development, with engineering responsibility for process definition and requirements management. Current development interests include software engineering, interoperable tools, and software process. Present address: TRW/Integrated Engineering Division, 12900 Federal Systems Park Drive, Fairfax, VA 22033.

Identifying Reuse Opportunities Within and Across Organizational Boundaries

Ronald L. Owens
Manager, Washington Operations
Systems and Technologies Division
CACI INC. - FEDERAL
1600 N. Beauregard Street
Alexandria, VA 22311-1794
Phone: (703) 824-4540
Fax: (703) 931-6530

Marrea H. Riggs
Director, Army Reuse Center
USAISSDCW
ATTN: ASQB-IWE-R Stop #4
Ft. Belvoir, VA 22060-5576
Phone (703) 285-9007

IDENTIFYING REUSE OPPORTUNITIES WITHIN AND ACROSS ORGANIZATIONAL BOUNDARIES

Ronald L. Owens, CACI, INC. - FEDERAL
Manager, Washington Operations

Marrea H. Riggs, USAISSC
Director, Army Reuse Center

Abstract - This paper introduces two demonstrated approaches for identifying internal (and external) reuse opportunities as a critical "first step" in systematic implementation of software reuse within organizations. When combined, these approaches support pursuit of short-term reuse benefits while incrementally establishing the reuse environment, or infrastructure, essential in achieving more significant long-term reuse benefits. Jointly developed by the Army Reuse Center (formerly, the RAPID Project) and CACI, INC. - FEDERAL, the first approach focuses on a formal domain analysis process that provides generic architectures, domain models, high-demand reuse categories, potential donor-client matchups and domain-specific reuse implementation plans in support of long-term benefits; this first approach continues implementation within the Program Executive Office for Standard Army Management Information Systems (PEO STAMIS). A second approach, initially developed in support of the Army Strategic Software Reuse Plan Task Force and now being implemented within the U.S. Army Information Systems Support Center (USAISSC), utilizes a less formal Reuse Technology Assessment (RTA) and high-level analysis techniques to provide initial guidance of organizational-specific reuse activities. In each approach, reuse cost equations are available to quantify the potential cost avoidance that can be achieved through opportunistic and systematic reuse. This paper provides an overview of each approach, identifies the objectives and primary products of each approach, and describes initial reuse benefits and current lessons learned from ongoing analysis activities.

Background

Our experience within the Army MIS domain has shown that one significant inhibitor to achieving significant results from reuse continues to be the lack of upfront planning and coordination; organizations fail to identify reuse opportunities and develop high-level strategies for efficiently implementing reuse, while minimizing the impact on program managers in terms of costs and schedule. Reuse often is addressed only after entering the final stages of the detailed design or commencing actual coding. Since it is often difficult to identify and evaluate specific reuse opportunities, specify when reuse benefits can be achieved, and quantify the value of those future benefits, senior managers are frequently unwilling to invest the time and resources required to implement a long-term reuse program.

Anticipating a Department of Defense (DoD) environment that will include smaller numbers of new systems, reduced funding levels and extended life-cycles for existing legacy systems, it is critical that we acquire the information to be able to focus limited resources to those opportunities offering the greatest potential for return-on-investment as a result of software reuse. Our experiences have identified a number of programmatic, social and technical inhibitors that must be systematically overcome if we are to achieve the promise of software reuse. We have seen that several of these inhibitors are the direct result of a lack of upfront planning; in failing to develop a high-level strategy for incrementally implementing reuse, there may be little opportunity to minimize the impact on program managers in terms of schedules, costs and risks. Developers will likely be given neither the opportunity nor the motivation to develop their requirements or complete their designs with planned reuse in mind. If addressed at all, reuse frequently will be considered only after entering the final stages of the detailed design or commencing actual coding, thereby reducing the potential returns-on-investment associated with software reuse throughout the Software Development Life-Cycle (SDLC).

We have developed and implemented a flexible, repeatable strategy for conducting a domain analysis, developing a domain-specific reuse implementation plan that incorporates five essential facets of reuse activities, and driving engineering activities in support of both short-term and long-term reuse benefits. Our reuse strategy takes advantage of the Defense Software Repository System (DSRS), an automated library system currently being used within Defense Information Systems Agency/Center for Information Management (DISA/CIM) and each of the individual Services, and documented procedures to guide the acquisition, evaluation, certification and reapplication of reusable components from each phase of the SDLC.[1]

Approach One: Formal Domain Analysis

Cohen and others state that we must identify both the processes and the products required to support reuse, and that we must continue moving from an *ad hoc* reuse approach to a systematic, incremental reuse process.[2] Unfortunately, in most cases, the identification of specific reuse opportunities, quantification of potential reuse benefits, and development of a specific strategy for implementing reuse can only be determined after thorough analysis within the particular domain or organization. A number of factors will determine both the degree of reuse potential and the time frame for achieving significant reuse within a particular domain. These include the number and size of systems within the domain, complexity and uniqueness of system requirements and interfaces, current SDLC phase of individual programs, choice of development methodologies and environments, and availability of reuse expertise and automated library tool capabilities. Our domain analysis process facilitates the collection and evaluation of these factors while providing an opportunity to develop an organizational-specific "blue print" to guide systematic, incremental and cost-effective implementation of software reuse.

The formal domain analysis approach described below has been implemented within PEO STAMIS and provides the basis of the current DoD Domain Analysis and Design Process Guidelines.[3] Key aspects of this proven, flexible and repeatable strategy for conducting domain analysis and reuse engineering activities in support of both short-term and long-term reuse benefits include:

- Identification and description of domain (and subdomain) boundaries,
- Initiation and support of one (or more) short-term Reuse Pilot Project(s),
- Development of generic architectures and domain-specific models,
- Implementation and population of a Domain Knowledge Database,
- Identification of horizontal and vertical reuse opportunities within the domain,
- Identification of generic and domain-specific high demand reuse categories,
- Quantification of potential short-term cost avoidance and long-term cost savings, and

- Preparation of a multi-year, domain-specific Reuse Implementation Plan.

Our goal has been to develop a flexible approach to reuse that can be tailored, in response to information gained through the domain analysis process, to maximize the benefits of reuse within the specific target domain. It was clear that our approach must have the ability to focus limited reuse resources on those programs that could provide the greatest short-term benefits from reuse, while still "setting the stage" for greater reuse payback in years to come.

To that objective, we have seen that it is essential that a reuse infrastructure be established to take advantage of domain analysis information and to guide the active pursuit of planned, systematic reuse at each phase of the SDLC.[4] The organization's policies, standard procedures and technical capabilities must be sufficient to drive the development of reusable products and consideration of reuse opportunities within, and among, individual programs (or product lines) within the domain. We have determined that critical aspects of the domain-specific reuse implementation plan must include domain analysis, indoctrination and training, internal reuse working group activities, domain implementation support, and the certification, population and utilization of the automated library system. It is clear that the domain-specific reuse implementation plan must have the goal of incorporating domain engineering activities (i.e., domain analysis, domain design and domain implementation) through the use of reuse libraries and other domain environmental support resources.

This approach was initiated within PEO STAMIS in early FY92 by the Army Reuse Center and CACI, INC. - FEDERAL. A challenging environment for software reuse, PEO STAMIS includes geographically distributed programs that utilize diverse development environments and currently exist at SDLC phases ranging from pre-requirements to fully fielded systems. FY92 reuse activities focused on the initial high-level analysis of the PEO STAMIS domain, detailed domain analysis and modeling of the Logistics Supply subdomain, and initiation of an FY92 Pilot Project to pursue short-term cost avoidance and to provide a "proof of concept" for reuse. FY93 objectives have included completion of a domain analysis within the PEO STAMIS Logistics domain, expanded reuse training, support of PEO STAMIS Executive Reuse Steering Committee and Domain Implementation Working Group (DIWG) meetings, and focused domain implementation activities within key development programs to achieve modest FY93 cost avoidance. More importantly, FY93 provided an opportunity to plan for coordinated reuse within PEO STAMIS by identifying specific reuse opportunities, both within and across PEO STAMIS domain boundaries, that could be achieved during FY94 and beyond.

One essential aspect of our analysis approach is the development of inter-related analysis documentation that provides a thorough description of the target domain, identifies and evaluates specific reuse opportunities, and recommends a detailed and domain-specific implementation strategy. A Domain Definition Report (DDR) identifies domain boundaries, defines principal subdomains, and determines relationships and links between the subdomains. The DDR provides graphic representations and narrative descriptions of logical, contextual, and behavioral domain models and a domain-specific generic architecture. A Reuse Opportunities Report (ROR) identifies reuse opportunities within the specific domain; the Reuse Opportunities Report includes a description of domain-specific High-Demand Categories (i.e., those types of components or requirements that will have the greatest reuse potential within the particular domain) and identification of specific donor-client matchups within the domain. A Reuse Implementation Plan (RIP) presents a multi-year strategy for implementing cost-effective reuse within the target domain. This domain-specific RIP addresses requirements for domain analysis, training and reuse engineering support, and quantifies potential cost avoidance that can be achieved through short-term and long-term reuse.

In addition, a Domain Knowledge Database is used to capture domain information and individual system characteristics in support of future development and maintenance efforts. The Library Population Report (LPR) recorded the results of the FY92 Pilot Project by identifying the number and types of components certified for inclusion in the library and by quantifying actual FY92 cost avoidance as a result of reuse between PEO STAMIS programs participating in the FY92 pilot project. Similar to the LPR, a Library Activities Metrics Report (LAMR) highlights FY93 activities by documenting the number and types of components certified for inclusion in the library, actual library extractions in support of the target domain, and actual and anticipated cost avoidance/cost savings.

While our approach to reuse is similar to others now being developed within DoD and industry, two factors warrant recognition. First, we have seen that the primary determinant in reuse success is the development and incremental execution of the domain-specific Reuse Implementation Plan. Second, while other approaches are being developed or initiated, our approach is already being implemented within the challenging and representative PEO STAMIS domain. The necessary infrastructure has been established; essential training is being conducted; short-term benefits already have been achieved; and systematic pursuit of long-term reuse, as recommended in the *PEO STAMIS Reuse Implementation Plan*, is in progress.

Approach Two: Reuse Assessment and High-Level Analysis

Recognizing that formal domain analysis activities can require significant investments of both time and resources, we have developed an abbreviated analysis process that utilizes a preliminary assessment of existing reuse capabilities and high-level evaluation of organizational-specific reuse opportunities to prepare an initial Reuse Implementation Plan. Currently being implemented within USAISSC, this streamlined process can be either a cost-effective "first step" in the overall domain analysis process or, in selected cases, a short-term alternative to the more formal domain analysis process. However, it is important to understand that the objectives of this reuse assessment and analysis process are necessarily more limited than those associated with a domain analysis. The more formal process used within PEO STAMIS focuses on identification of generic architectures, domain-specific models and high-demand reuse categories as essential interim steps toward the development of a very detailed Reuse Implementation Plan. The specific objectives of the more streamlined reuse assessment and high-level analysis process are to identify and evaluate reuse capabilities already available within the target domain, provide an initial evaluation of domain-specific reuse opportunities and develop an abbreviated Reuse Implementation Plan to guide near-term reuse activities.

This cost-effective approach has been developed to identify and, wherever feasible, take advantage of procedures, guidelines and other reuse capabilities already available to the target organization. The guiding principle of this approach is to identify and evaluate potential reuse opportunities in order to focus limited reuse resources to those reuse activities that offer the greatest potential for near-term reuse payback; the three primary activities in the Reuse Assessment and High-Level Analysis process are shown below:

- Conduct an organizational-specific Reuse Technology Assessment,
- Conduct a High-Level Reuse Analysis, and
- Develop and Initiate an organizational-specific Reuse Implementation Plan.

Preliminary organizational-specific information is acquired through conduct of the Reuse Technology Assessment (RTA) and one or more High-Level Reuse Analyses. This information supports the development of a Reuse Implementation Plan and, maintaining the near-term focus, can provide a valid starting point for one or more reuse pilot project(s) providing both proof-of-concept and short-term benefits, primarily based on opportunistic reuse.

The standard products of this approach will include a Reuse Technology Assessment Report and preliminary Reuse Implementation Plan. As conducted within applicable USAISSC Directorates and Software Development Centers, the goal of the RTA is to quickly identify, understand and document the reuse expertise, capabilities and experience already available within USAISSC. We have seen that it is important to understand the formal and informal reuse activities already being carried out within individual programs. This resident knowledge in software reuse will provide the first "building block" in the development of a cost-effective and systematic USAISSC-specific Reuse Implementation Strategy.

Continuing this process, we have determined that the subsequent High-Level Reuse Analyses (HLRA) are critical "next steps" in determining the appropriate scope, complexity and individual components of the organizational-specific Reuse Implementation Strategy. Within USAISSC, accurate knowledge of individual directorates and development centers (i.e., domain boundaries, functional or organizational overlaps, programming and environmental dependencies, and status of all current and planned systems) will provide the necessary "building blocks" for development and initiation of the USAISSC Reuse Implementation Strategy. This step will be essential in determining the composition of initial reuse working groups, identifying preliminary training requirements and focusing USAISSC reuse resources to those organizations or programs with the greatest potential for short-term and near-term reuse benefits.

As discussed in a recent paper presented at WAdaS'93,[5] the *PEO STAMIS Reuse Implementation Plan* has demonstrated that the systematic development and initiation of a organizational-specific reuse implementation strategy can set the stage for cost-effective, incremental initiation of software reuse activities within organizations. Whether developed by formal domain analysis or abbreviated reuse assessment and high-level analysis, we believe the Reuse Implementation Plan will continue to be an essential factor in successful and cost-effective reuse implementation. The following sessions briefly discuss five principal facets of successful reuse implementation, provide examples of reuse cost equations being used by the Army Reuse Center, and highlight interim benefits and "lessons learned" from our continuing reuse initiatives.

Principal Facets of Successful Reuse Implementation.

We have identified five "facets" of software reuse technology that have proven to be essential in developing and implementing successful reuse programs. In each of the approaches identified above, one of the critical results will be the development of a Reuse Implementation Plan that incorporates activities associated with each of the five facets that are appropriate to the specific organization. Referring to Figure 1, **Domain Analysis** activities provide the detailed analysis required to develop domain models and generic architectures, identify short-term and long-term reuse opportunities, identify high-demand categories, and quantify potential cost avoidance/cost savings. Experience has shown that establishment and active participation of **Reuse Working Group Activities** are vital to the success of reuse activities within and across organizational and/or functional domains. Looking again at PEO STAMIS, we continue to support two levels of working group activities, the Executive Steering Committee and a Domain Implementation Working Group (DIWG). **Training and Technology Transfer** provides initial reuse orientation and a range of essential training instrumental in achieving willing, enthusiastic and knowledgeable support of reuse activities. This facet of activities includes the investigation and, where appropriate, incorporation of developing reuse technologies. **Domain Implementation** provides analysis, software engineering and other dedicated technical support in the identification and acquisition of specific reusable components, integration and/or re-engineering of those components, and definition and design of new high-demand reusable components. These tasks are often essential in actually achieving the potential reuse benefits identified during earlier domain analyses. **Certification of Reusable Components** supports the evaluation, certification and maintenance of high-quality, high-demand Reusable Software Components (RSCs), and ensures both quality assurance and strict configuration management.

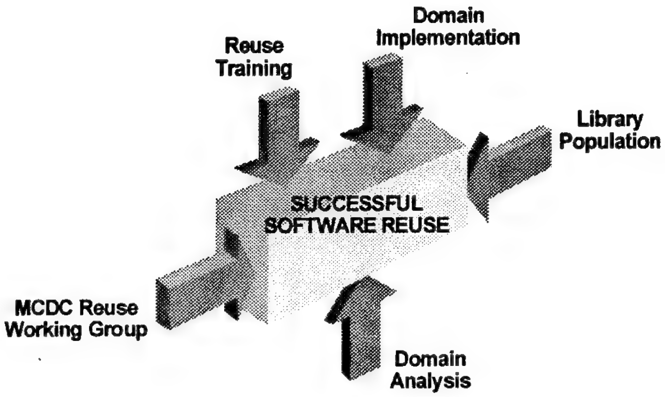


Figure (1): Multiple Facets of Successful Software Reuse

Reuse Cost Equations

Working with industry and the U.S. Army Cost Economics Analysis Center, the Army Reuse Center continues to develop, refine and validate reuse cost equations to quantify the benefits of cost avoidance and/or cost savings that can be achieved through opportunistic and systematic software reuse.

As a first example, the following equation is currently used to quantify the cost avoidance that can be achieved through reapplication of reusable Ada code components. Originally developed in support of reuse activities within the Army Strategic Software Reuse Plan (ASSRP) Task Force, this equation takes advantage of validated data points from recent Army, Navy and ARPA programs to provide ranges of potential (or achieved) cost avoidance:

$$BENEFITS = SLOC_R \left[C_{SLOC} - (C_{LIB} + C_{INT})\right] - C_{IRO}$$

For this equation,

$SLOC_R$ identifies source lines of code, including blank lines and comments,

C_{SLOC} provides the "raw benefits" (i.e., cost/SLOC for developing new code),

C_{LIB} recognizes the investment costs of identifying and installing code into the library,

C_{INT} recognizes the investment costs associated with integration of reusable code, and

C_{IRO} includes the incremental costs associated with identification of specific reuse opportunities.

It should be noted that Incremental Reuse Opportunities Identification Costs of a distinct reuse opportunity (C_{IROi}) is calculated by multiplying total Reuse Opportunities Identification Costs (C_{RO}) times the quotient of the source lines of code ($SLOC_{Ri}$) of the particular opportunity divided by total source lines of code ($TSLOC_R$) of all viable reuse opportunities within the organization.

$$C_{IRO_i} = C_{RO} * \frac{SLOC_{R_i}}{TSLOC_R}$$

The following paragraphs provide an example of the usage of these new equations. We will assume that a customer organization has a total FY93 Reuse Opportunities Identification Costs (C_{RO}) of \$120,000. Furthermore, we will assume that five specific reuse opportunities that could be achieved during FY94 have been identified:

Program A:	4000 SLOC
Program A:	3250 SLOC
Program B:	8500 SLOC
Program C:	12,150 SLOC
Program C:	10,500 SLOC

For this example, we assume the customer needs to evaluate the potential cost avoidance associated with Program B. The first step is to calculate Total Source Lines of Code ($TSLOC_R$) associated with all five opportunities:

$$TSLOC_R = 4000 + 3250 + 8500 + 10{,}500 + 12{,}150 = 38{,}400$$

We then can use the revised cost avoidance equation to calculate the range of potential cost avoidance associated with the reuse of 8,500 SLOC within Program B:

$$\textit{Cost Avoid}_{\textit{Prog B}}$$

$$= 8{,}500\ [65.00 - (1.34 + 6.57)] - [120{,}000 * 8{,}500/38{,}400]$$

$$= \$485{,}265 - \$26{,}562 = \$458{,}703\ \text{(High)}$$

$$= 8{,}500\ [65.00 - (1.34 + 26.00)] - 120{,}000 * 8{,}500/38{,}400]$$

$$= \$320{,}110 - \$26{,}562 = \$293{,}548\ \text{(Low)}$$

As shown in the example above, the Incremental Reuse Opportunities Identification Costs associated with Program B are \$26,562. The resulting cost avoidance to Program B should be in the range of \$293,500 to \$458,750.

Other reuse cost equations can project the potential cost savings associated with systematic reuse of common SDLC components or common functions within multiple programs. The following equation, originally developed in support of the Advanced Tactical Command and Control System (ATCCS) program, projects the cost savings associated with the reuse of common software (across the SDLC) among multiple functional users:

$$\text{COST SAVINGS} = [SLOC * SYSTEMS * C_{SLOC}]$$
$$- [SLOC_R * C_{SLOC} * C_{RD} * C_T]$$

The first half of the equation represents the total cost if each system independently develops the projected code; the second half of the equation calculates the projected cost if jointly developed (or developed by one system to be reused by other two programs). In the combined equation,

SLOC identifies the amount of code that would have to be independently developed if no reuse,

SYSTEMS represents the number of systems that require the code to be developed,

$SLOC_R$ identifies the amount of developed reusable code (from requirements to testing) in SLOC,

C_{SLOC} reflects the development cost/SLOC of new code from requirements to testing (for ATCCS, $200/SLOC),

C_{RD} consists of the design time factor per SLOC for maximizing the reusability of future code (for ATCCS, factor used was 1.15), and

C_T recognizes the time factor per SLOC added to the testing of reusable code (for ATCCS, factor used was 1.20).

As an example of its use, assume that three programs have agreed to jointly develop 150,000 SLOC of common software. In each case, code developed by one system will be reused, without modification, by the other two programs. If each program had developed the entire 150,000 SLOC independently, the resulting cost would have been:

$$COST_{INDEPENDENT}$$
$$= 150{,}000 \text{ SLOC} * 3 \text{ systems} * \$200/\text{SLOC} = \$90{,}000{,}000$$

Using the equation shown above, we can now calculate the estimated cost of jointly developing and sharing common software as shown below:

$$COST_{SHARED}$$
$$= 150{,}000 \text{ SLOC} * \$200/\text{SLOC} * 1.15 * 1.2 = \$41{,}400{,}000$$

The resulting cost savings would be the difference between the two alternatives, i.e., $90,000,000 minus $41,400,000, or a resulting cost savings of $48,600,000 among the three systems.

Interim Lessons Learned in Implementing Reuse

As the Army Reuse Center continues to guide and/or facilitate software reuse activities within PEO STAMIS, USAISSC and other Department of the Army organizations, we have observed a small number of "lessons learned" that are repeatedly validated by other industry and DoD reuse initiatives. Our experience has shown that each of these factors can be crucial in determining the degree of reuse success that we are able to obtain:

1. Develop and Implement a Domain-Specific Reuse Implementation Plan.

As discussed above, we have found the development and execution of a domain-specific Reuse Implementation Plan to be a crucial step in cost-effective and successful long-term software reuse. The Reuse Implementation Plan can provide an efficient, organizational-specific, multi-year strategy for incrementally pursuing the benefits promised by software reuse. Recognizing current programmatic and economic limitations, managers must frequently pursue short-term reuse benefits and return-on-investment while still investing in the domain analyses and technologies essential to achieving long-term reuse benefits. Taking advantage of guidelines, procedures and reuse capabilities already available within the organization, the Reuse Implementation Plan provides a domain-specific blueprint to guide incremental execution and periodic evaluation of domain-wide reuse activities.

2. **Ensure Management Commitment and Participation.**

A frequently noted observation which we have confirmed is that management commitment at all levels of the organization is essential if reuse benefits are to be maximized! We have observed cases, within both DoD and industry, where the commitment of senior management has been offset by the reluctance or actual resistance of middle managers or senior technical personnel. We have seen instances where technically-viable reuse opportunities, welcomed by both donor and potential client, have been negated by schedule, programmatic and/or political concerns. We have documented situations in which apparently viable reuse opportunities were negated because of implicit environmental dependencies and contractual risk issues associated with the reuse of "outside" code. As a result, we strongly recommend that organizational-wide reuse activities be guided and centrally-coordinated at the headquarters or executive management level. Key objectives will be the issuance of reuse policies, evaluation and prioritization of individual reuse opportunities, and, where necessary, resolution of conflicts. We have seen that Reuse Steering Committees that include active executive or headquarters-level participation can successfully identify and overcome many of the programmatic, political and social inhibitors to reuse.

3. **Pursue both Short-term and Long-term Reuse Benefits.**

It has been accepted that a systematic approach to software reuse offers the greatest long-term potential for increasing productivity and return-on-investment while reducing the time and costs associated with the development and maintenance of tomorrow's complex software systems. Reuse initiatives conducted in support of the Army Reuse Center and the DISA/CIM have reaffirmed that domain managers must invest in upfront planning and analysis if they are to achieve these goals. Organizations must develop a strategy and implement the policies and processes necessary to ensure that the domain-specific information contained in these resources are incorporated into future development projects from the very onset of initial planning. Our goal has been to develop a flexible and incremental approach to reuse that provides a viable and cost-effective solution for maximizing both near-term and long-term benefits of reuse within a specific organization or target domain. This approach provides the ability to focus limited reuse resources on those programs that provide the greatest potential for short-term reuse benefits while "setting the stage" for significant payback in years to come.

Using the *FY92 PEO STAMIS Reuse Implementation Plan*[6] as example, the short-term plan (FY93) focused on supporting independent reuse activities within individual programs; guiding reuse training and indoctrination across PEO STAMIS; facilitating periodic meetings of PEO STAMIS Executive Reuse Steering Committee and Domain Implementation Working Group (DIWG); populating the Army Reuse Center Library with high quality components that would be reusable within current and planned PEO STAMIS development efforts, and achieving modest FY93 cost avoidance (or return on investment). More importantly, FY93 was seen as an opportunity to conduct the domain analyses, provide training and indoctrination, and support intra-service and inter-service coordination that would be required to implement cohesive, long-term reuse plans within PEO STAMIS, Department of the Army and DoD. Numerous short-term benefits achieved during the first 10 months included the following:

- Initiation and completion of a FY92 Reuse Pilot Project that provided a return on investment of over 4-to-1;
- SAMS-I/TDA and SAAS-1/3 reused Ada code from the Army Reuse Center Library;
- Establishment of the infrastructure essential to achieving systematic reuse within PEO STAMIS, i.e., Reuse Steering Committee, Domain Implementation Working Group, reuse training and technology transfer;
- Development of domain-specific models and generic architecture in support of future reuse activities;
- Identification of the common system services and domain-specific functionality that will have the highest demand for reuse within the PEO STAMIS Logistics Supply domain;
- Identification and quantification of potential donor/client matchups that provide the basis for opportunistic and systematic reuse, within and across PEO STAMIS boundaries during FY94-FY95; and
- Development of a three-year PEO STAMIS Reuse Implementation Plan that concentrates reuse efforts on those programs offering the greatest reuse.

The longer-term reuse plan (FY94-FY95) takes advantage of the infrastructure and FY93 reuse investments to pursue systematic and opportunistic reuse within PEO STAMIS. This Reuse Plan focuses on coordinated and mutually-supportive reuse activities within PEO STAMIS, onsite reuse engineering support for key programs, continued evaluation and incorporation of developing technologies, and planned reuse within PEO STAMIS and across principal Army domains. Primary FY94-FY95 objectives would seek to institutionalize software reuse within PEO STAMIS, achieve quantifiable cost avoidance or cost savings, and populate the Army Reuse Center Library with high quality, highly reusable components from each phase of the SDLC.

4. **Pursue Both Systematic and Opportunistic Reuse.**

The majority of reuse initiatives over the last decade have focused on opportunistic (or ad hoc) reuse. Primarily reusing low-level algorithms and code components, the returns-on-effort in many cases have been minimal. As the technical inhibitors to reuse continued to be identified and resolved, we have seen the gradual change to a new focus on incremental, systematic reuse across the SDLC. Current literature praises the potential for systematic (or planned) reuse while minimizing the effectiveness or desirability of opportunistic reuse.[7]

While we strongly encourage this paradigm shift, our experience has shown that implementing and achieving systematic reuse is

[7] Opportunistic or ad-hoc reuse most often focuses on the reuse of existing components from commercial and DoD libraries. In many cases, reuse will be pursued at an individual or small-group level after projects are well into the detailed design or coding phases.. The availability of efficient reuse libraries and well-documented, fully-tested components become critical to opportunistic reuse. Systematic reuse focuses on planned reuse across the SDLC. Success requires early identification of opportunities and disciplined commitment to achieving those goals; systematic reuse often requires substantial upfront investments and may require significant revisions of an organization's existing procedures and development processes.

not easy. As mentioned earlier, there will frequently be significant investments required in terms of analyses, training and reuse capabilities. Significant cost savings and returns-on-investment associated with systematic reuse will often be years away. Numerous programmatic, contractual, technical and political inhibitors require resolution. While it is clear that systematic reuse will provide the majority of reuse benefits over the life-cycle, we have seen that opportunistic reuse can facilitate cost-effective pilot projects, support the validation and refinement of reuse processes, procedures and methodologies, and provide short-term cost avoidance to partially offset the investments required for systematic reuse.

5. Refine and Expand the Use of Reuse Economic Models.

We continue to observe the need for expanded capabilities to identify and evaluate alternative reuse strategies. Similar to the objectives of Business Case Analyses, an economic basis needs to be provided to justify reuse expenditures and quantify realistic reuse objectives. Looking at the *Army Strategic Software Reuse Plan*[8], a plan developed to guide the systematic implementation of software reuse within Department of the Army, numerous tasks highlight the criticality of selecting, refining and using reuse economic models to guide reuse activities within the MIS, Command and Control, and Embedded Weapon Systems domains. Domain and program managers need reuse economic models that can support the estimation of relative reuse potential (prior to initiation of time-consuming and expensive domain analyses) and quantification of potential cost avoidance/savings throughout the SDLC. In today's environment of reduced funding and stretched life-cycles, they need readily-available capabilities to focus limited resources to those opportunities that have the greatest potential for significant cost savings.

6. Advance the State-of-the-Practice in Automated Tools and Capabilities.

While today's technology is sufficiently mature to support both opportunistic and systematic reuse, we have identified numerous areas in which enhanced technology will be the key to greater effectiveness and increased returns-on-investment from reuse.

Current library systems require time-consuming searches by technical engineers already familiar with the particular library system. We need to provide better graphic user interfaces, utilize innovative display techniques and take advantage of artificial intelligence techniques to assist users in identifying, evaluating and selecting reusable components. Two areas that are currently very labor intensive and quite time-consuming are (1) initial analysis and modeling of the domain and (2) certification and maintenance of library components; we must implement automated capabilities that can help streamline these essential processes.

We must continue to provide increased interoperability among DoD and industry libraries; to this goal, DoD is already striving for intercommunications and interoperability among the Defense Software Repository System (DSRS), Central Archive for Reusable Defense Software (CARDS) and Asset Source for Software Engineering Technology (ASSET) libraries.

We need to identify and implement technologies that can support our paradigm shift in focusing on the reuse of requirement specifications and designs. We need the capabilities to link, display and track high-demand reusable components through the SDLC. We must expand our abilities to collect and use both quantitative and qualitative metric data, which is essential in evaluating the contents of libraries and the effectiveness of the libraries themselves.

Finally, we must continue expanding our view of reusable objects within the domain. Already pursuing the reuse of requirements, specifications, designs, code, tests and documentation, we need to develop capabilities to actually reuse our experience and inherent knowledge of specific organizations or domains. Current generational-type development systems that focus on the reuse of templates or data skeletons seek to utilize existing experience or knowledge. Those products whose primary objective is to re-engineer or transform existing systems provide others examples of methodologies that reuse existing knowledge.

BIBLIOGRAPHY

1. Owens, R., *"Reusable Ada Products for Information Systems Development (RAPID): One Solution to Planned Software Reuse"*, © SofTech, Inc., 5 November 1991.

2. Cohen, S., K. Kang, J. Hess, W. Novak and A. Peterson, *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Software Engineering Institute, CMU/SEI-90-TR-21, November 1990, pp. 2-6.

3. *DoD Center for Software Reuse Operations (CSRO) Domain Analysis and Design Process, Version 2,* SofTech, Inc., Contract No. 62K-RF029C, Document No. 1222-04-210/30.1, 30 December 1992, pp. 4-1 through 4-147.

4. *Army Reuse Center Support Across the Software Development Life-Cycle*, SofTech, Inc., Contract No. DAEA26-87-D-2001, Document No. PD167, 12 February 1993.

5. Owens, R., Development and Implementation of a Domain-Specific Reuse Plan, *Proceedings of the 1993 Washington Ada Symposium (WAdaS '93)*, pp. 31-42

6. *"PEO STAMIS Reuse Implementation Plan, Version 2"*, SofTech, Inc., Contract No. DAEA26-87-D-2001, Document No. 1213-65-210/6.1, 1 December 1992.

7. *Army Strategic Software Reuse Plan*, Office of the Director of Information Systems for Command, Control, Communications and Computers (ODISC4), 31 August 1992, pp. 47-54.

Ronald L. Owens
Manager, Washington Operations
Systems and Technology Division
CACI, INC. - FEDERAL
1600 N. Beauregard Street
Alexandria, VA 22311-1794
Phone: (703) 824-4540
Fax: (703) 931-6530

Mr. Owens is currently Manager, Washington Operations, Systems and Technology Division for CACI, INC. - FEDERAL. Co-recipient of the DoD Defense Standardization Program Outstanding Performance Award, he has been a leader in innovative reuse activities within Army, DoD and NATO. An active member of IEEE and ACM, Mr. Owens holds a Master's Degree in Computer Systems from the Naval Postgraduate School and is currently pursuing has PhD in Information Technology from George Mason University. His doctoral research addresses the development and validation of reuse economic models.

Marrea H. Riggs
Software Develop Center - Washington
Director, Army Reuse Center
USAISSDCW
ATTN: ASQB-IWE-R Stop #4
Ft. Belvoir, VA 22060-5576
Phone: (703) 285-9007

Ms. Riggs is currently the Director of the Army Reuse Center at the U.S. Army Information Systems Software Center, where she is responsible for managing the largest reuse center in the Department of Defense. A graduate of the Defense Systems Management College and a member of the Army Acquisition Corps, she is also a graduate of Victoria University in New Zealand and is currently pursuing a MS degree at Florida Institute of Technology. Ms. Riggs has more than 20 years experience managing the development and implementation of major automated information systems and has received numerous superior performance awards including the Superior Civilian Service Medal.

DEVELOPING REUSABLE OBJECTS IN ADA: A FIRST-TIME EXPERIENCE

William Schoen
IIT Research Institute
185 Admiral Cochrane Dr.
Annapolis, MD 21401

SUMMARY

IIT Research Institute, under contract with the Department of Defense (DoD), has recently completed a three year effort to develop reusable software components in Ada using object-oriented design (OOD). These objects will be used in the development of a large Army command and control system. The development focused on a number of characteristics of software that improve reusability. This paper describes some of these characteristics and the methodology used to produce the software. Some of the lessons learned during the development effort are also presented. In addition, the paper provides observations from the early stages of an effort to establish an in-house reusable software object repository.

INTRODUCTION

Although reuse of the various products of the software development process (analysis, design, code, documentation, etc.) has been discussed for many years, there has been a recent explosion of interest in this issue. With the advent of object-oriented software design methodologies, the potential for reuse has expanded. DoD support for a single primary programming language (Ada) has increased the potential for verbatim code reuse even further. Software reuse is also being encouraged by an increasing government-sponsored push for well-managed, easily-accessible libraries of reusable components.

As part of a multiyear project, our organization was tasked with developing scientific application software for a portion of a large Army command and control system. The software was to be developed as a set of reusable Ada software objects by re-engineering functionally designed baseline algorithms written in FORTRAN and C. In addition to the development of these application objects, the task included development of functional drivers to integrate the objects to perform specified functions. From the outset of the project, our goal was to produce software objects that could be reused on other object-oriented Ada projects throughout the company.

Our organization has a long history of experience with the reuse of software. Over many years, our organization has assembled a large library consisting primarily of FORTRAN subroutines. These subroutines have been catalogued and made available for reuse on numerous software development projects. The code is maintained by a permanent staff of software engineers, and is routinely modified and enhanced to support evolving requirements. We are now challenged with extending the existing reuse mechanism to facilitate and foster the use of object-oriented development methodologies and the Ada programming language.

The first part of this paper will discuss characteristics of reusable software and some of the particulars of developing reusable object-oriented code. Lessons learned from the development effort will also be presented. The second part of the paper will focus on establishing an environment that will promote effective reuse of object-oriented software.

DEVELOPING REUSABLE CODE

To promote the development of reusable code, strict software engineering standards were developed early in the design process, and were enforced by the project Quality Assurance (QA) Group during the review process. Of the many characteristics of software that improve reusability[1], our development effort primarily focused on readability, portability, and completeness. The significance of each of these characteristics is discussed below, along with a description of the methods used to develop software components with these characteristics.

Readability

Readability was considered important for two reasons. First, readability enhances the maintainability of the objects. Maintainability should be considered important whether or not code is intended for reuse. However, maintainability is particularly important for reusable objects because they are generally intended to have a longer life span. They are also expected to be used in multiple applications.

The second reason readability is important gets to the heart of the reuse objective. The benefits from reuse do not come from the *development* of reusable code, but rather from the *use* of reusable code. Developers cannot be expected to reuse code if they cannot understand its purpose and the way it behaves. Readable code is easier to understand.

To improve the readability of the code, developers were required to use meaningful variable names. Abbreviations were not permitted unless they were generally recognized in the problem domain. Procedure and function names were required to be as English-like as possible. Named parameter association (i.e., explicitly relating actual parameters to the corresponding formal parameters) was mandated except in the case of math functions and certain Text_IO procedures. For example, a procedure declared as follows:

```
     procedure Exchange (First       :
in out Integer_Type;
                         And_Second : in
out Integer_Type);
```

would be called as follows:

```
     Exchange  (First              =>
High_Number,
                And_Second          =>
Low_Number);
```

to exchange the values of variables High_Number and Low_Number of type Integer_Type. Deviation from these rules was rarely accepted, and only after obtaining project QA approval.

Portability

Portability enhances object reusability for obvious reasons. Ideally, the software should be designed for reuse under any operating system and on any platform. In this particular project, portability was essential. We were designing and implementing application objects that were intended to be integrated into a larger system by another contractor. Because the time for completion of the entire system was critical, we were tasked to start our portion of the effort before the final integration contract could be awarded and before some of the system requirements could be specified. Among the requirements that were not specified when we started our effort were the target platform, operating system, compiler, and database.

To insure portability, care was taken to avoid the use of implementation-dependent Ada features. For example, a set of primitive numerical datatypes with specified ranges and levels of precision was defined to avoid any reliance on Ada's INTEGER and FLOAT types, which are implementation-defined. Also, because Ada compilers are not required to implement UNCHECKED_DEALLOCATION, a memory management package was retrieved from a government-sponsored software repository and used to manage memory for objects that relied on dynamic memory allocation.

Every effort was made to keep objects independent of input/output and other system dependencies. In cases where this was not practical, the developers created *externals* (a term coined by our project). These externals were separated from the objects, and developed as independent Ada compilation units (i.e., packages or subprograms) with system-independent specifications. All dependencies on the operating system or data files were confined to the bodies of the externals.

One example of a component with an external dependency is Geodesic_Path_ Class, an object created to represent a physical path between two locations on the surface of the earth. Among the responsibilities of the object is the calculation of the azimuth relative to north for a particular geodesic path. Calculating the azimuth relative to *true north* is a purely mathematical exercise. Calculation of the azimuth relative to *magnetic north*, however, is dependent on a set of data that is updated periodically to account for changes in the position of the magnetic pole. To isolate Geodesic_Path_Class from direct dependency on a data file, a procedure designed to accomplish the data retrieval was developed as a separate compilation unit. As a result, the geodesic path object can be ported from one system to another without change, as can the specification of the external procedure. Only the body of the external procedure must be changed to accommodate a new environment.

Completeness

An object is considered *complete* if it provides all of the primitive operations that a client of the object might reasonably expect. In cases where the object is an abstraction of a tangible entity in the real world (e.g., a display terminal), this set of operations might be derived from a list of things the object is expected to do and have done to it. In the case of an object that provides a traditional data structure, such as a linked list, the set of operations are derived from a list of the ways in which the data structure will need to be manipulated. However, building a complete object often requires extending the capability of the object beyond the requirements defined by any one application. For example, although a particular application may not require that a linked list have a *Clear* operation, the reuse potential for a linked list is greatly increased if the *Clear* operation is included.

The nature of our effort made extending the functionality of objects a necessity rather than a luxury. Although the design process we used was very similar to a design process geared to the development of a fully integrated system, the objects were being developed to support a larger system which would be implemented separately with functionality in problem domains broader than our own. Although our requirements included the development of functional drivers to show how the objects could be used, it could not be assumed that our objects would be used in the larger system the same way our functional drivers illustrated.

The higher level objects we developed were identified by an analysis of our problem domain. In this case, the problem domain was reverse-engineered from the existing baseline software. Objects were identified that could be used in an object-oriented solution to satisfy functional requirements met by the baseline software. Additional lower level objects were identified in a subsequent series of design iterations following a specific design methodology.

Once an object was identified, development of the object was assigned to an individual member of the Software Engineering Group. Once the object was assigned, members of the Software Engineering Group, the Quality Assurance Group, and the Testing Group viewed the object largely as a stand-alone component. The developer was expected to build an object that not only satisfied the initial requirements, but also met other anticipated requirements, as was illustrated by the linked list example presented earlier.

DEVELOPING REUSABLE CODE: Lessons Learned

Setting Standards

In general, it is important to set standards and adopt a style guide early in the development process. It is equally important, particularly in a first-time effort, to be flexible and receptive to change. Developers and reviewers must recognize the difference between standards, which require strict adherence, and guidelines, which can be more loosely applied. Flexibility is important because experience will probably dictate a shift in the boundary between standard and guideline.

Readability

Readability must be supported by a clear set of standards and guidelines. Developers may grumble at first, but if the requirements are reinforced during code reviews, following the standards will quickly become a matter of habit. If a pretty-printer is available, it can be used to automatically format code to bring it into compliance with the style guide. However, it may be necessary to define the guidelines around inflexibilities inherent in the pretty-printer. For most developers, a pretty-printer that creates an output that must be tweaked to meet a standard is more of an annoyance than a help.

Readability standards and guidelines delivered an added benefit. Problem domain experts who had little programming experience, and no knowledge of Ada, were able to review our code. This resulted in better communication between developers and domain experts, which proved invaluable.

Portability

It is easy to become too complacent about portability when developing in Ada. Ada is widely touted as a well-defined, highly-stable language. This does not make portability automatic. Developers must become familiar with the implementation-dependent aspects of Ada. Coding standards must be established early to steer developers clear of the idiosyncracies of a particular compiler. If practical, code should be compiled under more than one compiler. Object test drivers should be compiled and executed on more than one platform. Familiarity with more than one compiler highlights some of Ada's implementation dependencies, particularly differences in the definition of the predefined integer and floating point types.

Completeness

Because objects were treated as stand-alone components soon after they were identified, it was easy to extend the functionality of the objects to make them complete. In the absence of good judgement, however, this can be perceived as a license to over-design. Most objects are intended to be an abstraction of the real world. This abstraction is limited, to some extent, by the significance of that object in a specific context. Completeness must be defined relative to that particular abstraction. For example, the abstraction of a display terminal for one problem domain may be little more than an on/off switch. For another problem domain, it may be a complex aggregate of circuits and controls. To construct an object suited to the second domain when developing for the first domain is clearly a case of over-design. In short, it is a mistake to develop objects that are intended to be all things to all applications.

SETTING UP A SOFTWARE REPOSITORY

As mentioned earlier, reuse is not new to our organization. An extensive library of software has been developed and maintained over many years. The tasking to produce reusable objects in Ada has provided wider opportunities for reuse throughout our organization. Consequently, a small group of developers from the project has begun to explore ways to extend the existing reuse mechanisms to promote reuse of the Ada objects, as well as other available products (e.g., design, code, algorithms, documentation, etc.) resulting from the project. The ultimate goal is to add a networked repository of reusable object-oriented Ada components, to develop methods of predicting costs and savings associated with reuse, and to provide incentives for effective and efficient reuse of the components. Although this effort is in its early stages, it may be useful to provide the following observations.

Impediments to Reuse

Impediments to reuse have been discussed at length in the literature. One impediment that is often mentioned is the Not-Invented-Here Syndrome. This refers to a tendency among software developers to reject without evaluation software developed outside their own organization. It should not be assumed that no similar impediment exists for software reuse within an organization. In fact, the Not-Invented-By-Me Syndrome can be just as big an impediment as the Not-Invented-Here Syndrome.

There are any number of reasons for a software developer to avoid using software developed by someone else. One reason is lack of trust in the quality of the software. Another is the recognition that there is risk associated with investing time into evaluating software written by someone else. An experienced programmer may well be able to accurately predict how long it will take to develop a piece of code from scratch. The time spent evaluating a software component that in the end could prove inadequate or inappropriate will be perceived as nothing but a waste of time.

Software developers may also choose to reinvent the wheel if they see the endeavor as interesting and a challenge. They may, on the other hand, be motivated to reuse code if reinventing it appears to be difficult and tedious, or if there is a system to recognize or reward them for reuse. Motivating developers to reuse code is often as difficult as motivating them to put in the additional effort required to develop reusable code. It probably requires some yet undiscovered combination of the carrot and the stick. Like the rest of the software development community, we are working to discover an effective combination.

Economics of Reuse

Many managers are waiting for evidence from across the industry to support their suspicion that reuse pays. We (and the industry) are seeking a proven economic model to help predict time and cost savings from reuse. The development of reliable models certainly depends on an accumulation of reuse experience and a standard set of reuse definitions. The literature is full of reuse statistics, but it is not clear that all authors mean the same thing, even when presenting data as basic as "percentage of reuse."

The economics of reuse are often oversimplified. Reuse is not free, even when the code is obtained without charge. Responsible software developers will not reuse code in their applications without thorough evaluation unless they have a high degree of confidence in its quality. Final responsibility for the performance of reused code (or the adequacy or accuracy of reused design and documentation) rests with the developer who uses the code, not with the developer who originally wrote the code. There is a cost associated with locating a software component that is a candidate for reuse. Also, the cost of analyzing a component to understand it and determine if it is appropriate is a cost that cannot be ignored.

The risk associated with embracing software reuse is greater for small-scale software development efforts than for large-scale, multiyear programs. Smaller projects do not have the luxury of averaging out costs and benefits over time and across program increments. Economic models for reuse will have to be relatively precise before they can be safely applied to smaller software development efforts. It may be unreasonable to expect a model that is sufficiently precise for smaller projects. In the absence of sufficiently precise models, managers and software engineers may have to rely on experience to provide them with a "feel" for the benefit obtainable from reuse. Developing that kind of experience takes time. The need for that kind of experience will delay full support for reuse.

Don't Bite Off More Than You Can Chew

It is easy to underestimate the effort required to set up an in-house software repository. Discussions within our organization started out with a decidedly idealistic tone. A great deal of time was spent considering what kind of standards software must meet before being accepted into the library. It was decided that a multilevel accreditation scheme would be necessary. Some code would be made available with a "Use at your own risk" advisory. Code that had been fully tested, or had a long history of successful usage would be given the library's seal of approval. The group initially decided that an elaborate tracking system should be set up to make the collection of metrics possible, and to allow the librarian to notify users of the software of correction and enhancements made to the code. Before long, the preliminary design of the in-house repository had all of the bells and whistles of the larger government-supported repositories. At that point, it became clear that it was not practical to immediately implement the library design in its entirety. In designing any system to support software reuse, the cost of the effort must be justified by the payoff. Therefore, an incremental development approach was adopted.

When considering the development of a reuse library, it is important to maintain a clear focus on the goal. The creation of a library is not the goal. The goal is to provide developers with better access to the products of previous software development efforts (algorithms, code, design, documentation, etc.). Anything that will help them avoid reinventing the wheel is better than nothing. Something as primitive as an index and a set of files that can be accessed over a local area network will provide some of the reuse support provided by a sophisticated online library. It is far from ideal, but it is a start.

CONCLUSION

Many challenges face an organization that decides to invest time and effort into promoting software reuse. Although there has been a great deal of recent exploration of this issue, much of the territory is uncharted. Software reuse undoubtedly has both hidden costs and hidden benefits yet to be discovered. To get the greatest benefit from reuse efforts, it is important to record the lessons learned along the way and to pass them on to other projects. Like all other organizations involved in software development, IIT Research Institute faces the continuing challenge of improving the mechanisms that promote sharing of both project experience and the more tangible software development products.

REFERENCES

Booch, G., Software Components with Ada: Structures, Tools, and Subsystems, pp. 34-35., The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

ABOUT THE AUTHOR

William Schoen is a Software Engineer with IIT Research Institute in Annapolis, MD. He is currently working on a large Army command and control system development project. William earned a bachelor's degree in Electrical Engineering from the University of South Florida and is currently pursuing a master's degree in Electrical Engineering from Johns Hopkins University in Baltimore, MD.

INTEGRATING Ada INTO REUSABLE SOFTWARE ENGINEERING

Huiming Yu and Joseph Monroe

Department of Computer Science
School of Engineering
NC A&T State University
Greensboro, NC 27411

Abstract

This paper describes a course which integrates Ada into software reusability training for undergraduate students. Our approach is to teach Ada, emphasize the features of Ada, and focus on reusable designs, reusable data structures, reusable components and reusable subsystems. An essential element for success is that we choose appropriate projects and give the students guidance in developing object-oriented systems in a progressive method. During the development process, students learn reuse driven methodology, analyze the features of Ada for reusability, and apply suitable features to each step.

I. Introduction

Software reuse has become a keystone in many current efforts to increase software development and maintenance productivity as well as the quality and reliability of software. The importance of software reuse is widely recognized by industry and government. Ada is a general purpose programming language which was developed for large, complex systems and is appropriate for a wide range of applications in the commercial and research communities. Commercial acceptance of the Ada programming language has been impeded by a lack of Ada trained programmers. Universities have traditionally played a critical role in satisfying both industry and government's demand for students fully trained as software engineers familiar with programming languages. In order to satisfy the requirements of industry and government, we developed an undergraduate course that integrates Ada into reusable software engineering.

Since most undergraduate students have only a little experience in programming, it is a non-trivial job to teach Ada and integrate it into software reuse. Questions, such as how to teach software reuse and combine it with object oriented design; how to teach Ada and combine it with developing reusable components and subsystems; as well as how to solve the problems the students have and combine them with the projects; etc, must be handled properly. Based on our experience, our strategy is to consider problems as they are encountered, and provide an appropriate approach to solve them. All these issues will be addressed in the following sections.

This paper contains four sections. Section two introduces how to teach the concept of reusable software engineering and Ada. Section three describes how to combine the teaching topics with developing the projects in a progressive method. Section four summaries our experiences and conclusions.

II. An Integrated Approach

We teach this course by first introducing reusable software engineering concepts, then teaching the Ada language and emphasizing its features for reusability. By combining each with related projects, students integrate Ada into reusable software engineering.

Teaching Reusable Software Engineering

The purpose of teaching reusable software engineering is to propagate the concept of reuse, cause the students to recognize the importance of reuse, learn reuse driven methodology, and apply it to design software systems. We teach this course to answer questions such as what is software reuse, why reuse, and what is required to implement software reuse technology.

This work is supported by CECOM Center for Software Engineering, Department of the Army, Fort Monmouth, NJ 07733-5000.

Introducing what is reusable software engineering is our first step. After this initial introduction, most students understand that reuse is the use of previously acquired concepts and objects in new situations. The problem we face is that some students think reusable software engineering only means to reuse code. We emphasize that reuse has a broader meaning that includes reusing data, data structures, designs, components and subsystems etc. At the same time, we teach the students to recognize that today's software development approach is to share work where individual developers design different components of the application. Software engineers must keep reuse in mind from the beginning of the system development so that they can provide reusable information and objects.

When emphasizing reuse, we answer the students' question why reuse. The students are divided into a few groups to discuss this topic and give their opinions. We sum up the students' opinions and point out that the benefits of applying reusability to software development are reducing the cost of software development, improving software quality, accelerating software production etc. Then we guide the students in discussing the questions, what is the cost of reuse and how to achieve a trade off between reuse and cost. Thus, the students move one step closer toward understanding reuse.

Guiding the students toward learning reuse methodology is another goal in this course. We introduce reusable patterns, reusable building blocks as well as other methodologies to the students. We combine these with the features of Ada and focus on the approach of reusable building blocks. We explain that one approach developers can apply to the development of massive, software-intensive system, is to construct small components whose behavior is well understood and then use them as building blocks for more complex systems. We emphasize that each component must be designed and documented well; otherwise it is useless. A few examples are used to explain these ideas as well as to help students understand reuse information and reuse objects.

After the students grasp what reuse is, why reuse and how the technology is to be used, they are asked what is the duty of computer science students. Most of them think they will be programmers or programming analysts after graduating and will be involved in many software system developments, modifications and maintenances. They recognize that these jobs require students to have excellent design ideas and good programming style and skill. In other words they must keep object oriented design in mind and adopt reuse driven methodology.

Teaching Ada

Ada is a specially designed language that provides many features to increase software productivity and reliability. Perhaps most important, Ada embodies and enforces the modern software engineering principles of abstraction, information hiding, modularity, and locality. It also provides a number of features that strongly support software reusability. Therefore it is a suitable language to be adopted for teaching reuse. We stress its features, integrate them into software reuse, guide students to design reusable components and subsystems, and to develop object oriented systems.

We teach students to design reusable components by using the generic program unit. The generic program unit is a new concept and a useful structure to support reusability. It is a template for a module and may be parameterized allowing reusable software components to be captured by a single generic definition. Since a generic program unit is totally different from traditional parameterization mechanisms that are usually restricted to variables, we teach it combined with examples. We introduce its concept by examples such as STACKs and LISTs while explaining the basic principles. Some students feel puzzled about a program unit working with different data type, We demonstrate how to use a STACK with various data types to help them fully understand the generic program unit. Finally students are required to develop a reusable component - QUEUE by using generic and package structures. By implementing a QUEUE, they not only understand generic unit well, but also become familiar with how to design reusable components.

Another topic is how to use the features of Ada to develop a reusable subsystem. Ada's package mechanism, subprograms, together with its generic units provide a rich vocabulary for expressing the modules of a system and their behavior as well as how they interface with one another. We teach the package combined with designing reusable subsystems. A WINDOW_MANAGER is an example of a reusable subsystem which consists of a few packages, such as IMAGE package, WINDOW package and SCREEN package. We adopt WINDOW_MANAGER as an example to analyze its structure, then point out that a subsystem is a collection of objects that work together to serve some common abstraction. Based on the students' knowledge we lead them to consider examples of reusable subsystems and analyze the reusable possibility separately.

Students are instructed to discuss the advantages of separate compilation and how it supports software reuse. They point out that separate compilation permits that reusable program units to be compiled separately as well as the creation of program libraries. In addition they note that later the application programmers can search program libraries to find suitable components and subsystems to be reused. Students illustrate these advantages with examples such as mathematical libraries and common utilities etc. Additionally, they are willing to have their own reusable components entered into a library.

By learning Ada the students reach the conclusion that Ada has many features to strongly support software reuse, and is a suitable language to be used for reusable software engineering.

III. A Project That Works

We train the students in this course by teaching, discussing, and developing projects. The projects are chosen to exemplify Ada and at the same time parallel the projects in object-oriented design. A few small projects are assigned and students are required to use object-oriented design, adopt reuse driven methodology, and implement them by using Ada. As an example to show the ideas and the methods, we use the Student Enrollment System. The Student Enrollment System is a relational database which includes basic student information, registration state, financial information etc. During the development process, the students play different roles.

Students As System Designers

Since most computer science students will be software engineers in the near future, they should be trained as object-oriented designers. They must recognize that object oriented development is an approach to software design and implementation in which the decomposition of a system is based on the concept of objects. They must understand the importance of using reuse driven methodology and implementing a system according to the requirements.

We divide the students into groups and assign a project to each group. They are required to use object-oriented design, adopt reuse driven methodology, and implement it by using Ada. We explain what is good programming style and require them to develop the assigned project in a very uniform and consistent style of programming that makes applications remarkably easy to modify even by other students who are not involved in the original project. First the students read the requirements carefully, design data structure, develop the system, then decompose the system into a few objects. In the first step they divided the system into a database dictionary, a report generator, a command subsystem, as well as an input subsystem. Based on reusability, they go a step further to decompose the report generator into report and report_interface, command subsystem into command_interface, sort&find, editor, and calculation, input subsystem into input_interface and information.

Students As Implementers of Reusable Components and Subsystems

Choosing appropriate structures to implement designed system is part of the training in this course. First the students identify which components and subsystems are reusable, then choose suitable structures, and finally build basic blocks. The following is an example of the student's program to implement a bubble sort algorithm by using generic and package.

```
generic                                    --implement sort
  type  element is private;
  type index is (< >);
  type array_type is array (index range < >) of element;
  with function "<'(element1, element2 : in element) return
boolean;

package sort is
procedure generic_sort(list: in out array_type);
end sort;
```

They use the following procedure to instantiate **generic_sort**.

```
package use_sort is
procedure usesort(list: in out listtype);
end use_sort;

package body use_sort is
procedure usesort(list: in out listtype) is
 function by_lname(element1, element2:    studentrec) return
boolean is
 begin
 return element1.lname < element2.lname;
 end by_lname;
```

```
                    ...
package lname_sort is new sort(studentrec, positive, listtype,
by_lname);
 begin
                    ...
          lname_sort.generic_sort(list);
                    ...
 end use_sort;
```

Another reusable component that students developed is **find**. They also use generic and package.

```
generic                 -- implement search
  type key is private;
  type element is private;
  type index is (< >);
  type list is array (index range < >) of element;
  with function equal (elem1: in key; elem2 : in element) return
boolean;

package find is
  function generic_find(fkey: in key;in_item: in list) return index;
 no_found: exception;
end find;
```

After the students implement sort and find, we lead them to construct a **sort&find** subsystem that is a common block in database systems and can be composed by **sort** and **find** packages. The purpose of this step is to guide the students toward building a new subsystem by using existing components. Through this step most of them found that building a subsystem based on reusable components is very easy, because they only need to write a few line of codes to glue them together.

By developing reusable components and reusable subsystems, the students learn how to design and implement basic components, construct new components by laying them on top of existing components. Also they learn to write good documentation for these reusable objects.

Students As Clients of Reusable Objects

The students play the role of a client of reusable objects in the last step. The purpose of this step is to train them to use previous designs, data structures, components and subsystems, and enforce software development with reuse.

The students are required to develop a database that records all customer related information such as customer name, address, telephone number, purchases, prices etc. for a Computer System Technology Incorporation. We require the students to fully utilize information and objects that are developed in the Student Enrollment System. They exchange source codes and documentation of the Student Enrollment System between groups, then identify which parts are reusable without modification, and which parts can be reused with little modification. By developing the second database, the students learn how to use existing idea and knowledge, as well as particular components and subsystems that have been developed previously.

IV. Conclusion

In this course we teach Ada and integrate it into software reuse, and achieve our goal completely. Our methodology is to combine software reuse into Ada teaching, design special projects to cover each topic, and guide the students to develop these projects in a progressive method. By taking this course, the students learn Ada, understand different levels of reusability, and use Ada to develop object-oriented systems. Also they learn how to design reusable components and subsystems, how to write the code on top of existing code, how to use existing ideas, knowledge, components and subsystems to build new systems.

References

[1] Booch, G. and Bryan, D., Software Engineering with Ada, *The Benjamin/Cummings Publishing Company, Inc.*, 1994.

[2] Booch, G., Software Components with Ada, *The Benjamin/Cummings Publishing Company, Inc.*, 1987.

[3] Freeman, P., "Classifying Software for Reusability", *IEEE Software*, January, 1987.

[4] Ichbiah, J. D., "On the Design of Ada", *Information Processing*, 1983.

[5] Krueger, C. W., "Software Reuse", *ACM Computing Surveys*, Vol.24, No. 2, June, 1992.

About the Authors:
Dr. Yu is an Assistant Professor of Computer Science at NC A&T State University. Her teaching and research interests are in the fields of software engineering, software reuse and concurrent programming languages. Dr. Yu may be reached by e-mail as cshmyu@garfield.ncat.edu.
Dr. Monroe is the Chair and Professor of Computer Science at NC A&T State University. His teaching and research interests include software engineering, object-oriented design and software reuse. Dr. Monroe may be reached by e-mail as monroe@garfield.ncat.edu.

A THREE–STAGE APPROACH FOR OOP EDUCATION

C. Thomas Wu
Naval Postgraduate School
Department of Computer Science, Code CS
Monterey, CA 93943

Summary

The paper addresses issues related to teaching object-oriented programming to beginners who have either some or no prior experience in computer programming. The three-stage, bottom-up approach is introduced and explained. With this approach, beginners learn object-oriented programming in a linear, incremental progression of non-OOP (Stage 1), semi-OOP (Stage 2), and full-OOP (Stage 3) stages. How the proposed teaching methodology is adapted to individual languages such as C++ and ADA is also discussed.

Introduction

As more and more disciplines of computer science are embracing object-oriented concepts, it is now a must to teach object oriented concepts to the students of computer science as early as possible in their studies. In addition, there is a strong need to re-educate those who studied computer science before the age of object-oriented paradigm. In this paper, we will present a proposed approach in teaching object-oriented programming to beginners (those with no programming or very little programming experience). Our objective in teaching OOP to beginners is to provide them with a solid foundation so they can embark on learning

- different object-oriented programming languages,
- different object-oriented analysis and design, and
- advanced OOP concepts.

In other words, we want to transform beginners into good object-oriented thinkers so they will be well prepared to absorb additional information to become true object-oriented programmers.

Proposed Approach

Our proposed method for teaching OOP proceeds bottom-up in three major stages with each stage subdivided further into smaller steps as necessary to adjust to the different capabilities of beginners. The three stages are as follow:

Stage 1: Non-OOP

In the first stage, we explain and illustrate the mechanics (syntax and semantics) of the chosen language. There will be no discussion on object-oriented concepts per se, i.e. we do not mention inheritance, polymorphism, etc. Notice that it is not our objective to teach any particular language, and therefore, we should be careful not to over-explain the language. In other words, we do not want to teach every detail of the chosen languages. We should keep an introduction to the language syntax and semantic at a minimum. Any additional (and necessary) languages features can and should be introduced in the later stages.

A small non object-oriented sample program is built. It is critical to use a unified sample program for all three stages, so the beginners can see how the program is improved by incorporating the OOP concepts. This will give a better appreciation to intermediate beginners on how and why OOP helps in building better programs.

Stage 2: Semi-OOP

In the second stage, we introduce the notion of code reuse. We improve earlier sample programs built in Stage 1 by (re)using the system supplied objects. At this stage, we do not create our own objects. We only use the existing predefined objects. The objective of this stage is to show how easily a program can be developed if we can reuse existing objects. A notion of object creation (by this we mean defining a new object class, not creating an instance of the existing class) is introduced in the next stage. Too often OOP courses (and books) introduce the notion of object creation before really teaching why we create objects. Only after learning and appreciating the power of reusing objects, one is truly ready to master the mechanism of object creation. This separation of object creation and object reuse allow us to teach the concept and importance of abstraction, encapsulation, and polymorphism in software development in this stage.

We avoid using the term "class library" to describe the system-supplied objects. A class is simply a mechanism to define its instances (i.e. objects), and as users of objects, we really do not care how they are created. All we care is that the objects behave as promised. We think the notion of "class" is explained too early by others. It is our experience that beginners cannot grasp the difference between "class" and "object". Also, by de-emphasizing a class, a classless (e.g. delegation) object-oriented system can be explained better latter.

Stage 3: Full-OOP

In the third stage, we finally introduce the full concepts of OOP. The concept of inheritance is introduced in this stage. The emphasis in this stage is the server programming, i.e. how to create a programmer–defined objects. The concepts of class and inheritance are explained and illustrated. Limitations and weaknesses of programs developed in Stage 2 are analyzed. The program is then improved by creating and using the programmer–defined objects. The importance of OOP to a team programming and the management of class library are other important topics to be covered in this stage.

An object-oriented design guideline is introduced in this stage also. This guideline is intended to provide an informal guideline for beginners to follow in designing programs. With this guideline, objects are classified into four categories of *Interface, Controller, Storage, and Application Domain.* Each category of objects are partitioned into three layers. The bottom layer is the system layer where the objects are system supplied. The middle layer is the domain layer. Objects in the domain layer are built from the system layer objects. Objects in this layer are not as general as those in the system layer. They support a higher abstraction that is little closer to the application areas. The topmost layer is the program layer. Objects in this layer are those specific to a program being developed. Degree of reusability decreases as you move up the layer. So the objects in the program layer are not expected to be reused in the other programs, while the objects in the system layer are expected to be reused in almost all programs. This four category, three layer paradigm is not intended to be a new OO design methodology. Rather, it is intended to be a guideline for introducing beginners to design methodology. This guideline complements different design methodologies.

Implementation Issues

There are several issues we need to resolve in actually using the proposed approach to teach the beginners.

Language

The first issue is the choice of object-oriented programming language. We would like to use a language that would enhance the learning process. We definitely have to avoid those languages with arcane and overly complex syntax and semantic. The ultimate goal is of course to make beginners become true object-oriented programmers, so we want a language that is least intrusive and domineering.

Beginners need a lot of hand holding at first. People learn best if they are given small doses of new concepts gradually. It is important not to overwhelm them with too many new concepts all at once. The pure object-oriented language such as Smalltalk may have an elegance and uniformity in treating everything in a program as objects, but this elegance and uniformity often overwhelm and confuse the beginners. One reason is because everything is an object in a pure language, we must design new kinds of objects in writing a program. In other words, Stage 2 programming is strictly speaking not possible with a pure language, but the mechanics of creating a new class and the decisions when to use the existing class or create your own is too difficult a concept for beginners to understand early in their studies.

Class libraries

The second issue is the use of class library. In order to teach a faster development and reusability, we need to have a good class library. Without any existing objects, the benefits of OOP cannot be taught easily to the beginners. But which class library should we use? The difficulty is that there are many different class libraries and they all support different paradigms. If the beginners are taught exclusively with one class library, they may have a hard time using a new class library. Just as we would like to minimize the effect of a chosen language, we would like to minimize the effect of a chosen class library. We want beginners to learn OOP, not how to use a specific class library.

Programming Tools

The third issue is the programming tools. People learn best by actual experimentation. In order to make the learned concepts really sink into the minds of beginners, we need some kind of programming tools that will illustrate, enhance, reiterate, and exemplify the concepts in a concrete form. It would be ideal if we could have an extensible programming environment, where a simple environment is provided for a real beginner, then a little more powerful environment for the second stage beginner, and so on.

OOP Paradigm

The last issue is the design methodology. After acquiring fundamental concepts of OOP, the students must learn some form of design methodologies. Teaching design methodologies immediately to beginners are too much; they will get confused. It's simply too early for the beginners. They probably do not have enough foundation to really appreciate the importance of design methodology in creating programs. What we have to do is to prepare them so they can absorb a new design methodology without too much problem. Here again as before, preparation should not limit them to a specific design methodology. Ideally, what we teach as prerequisite should be adaptable to any object-oriented design methodology.

Case Study

In the series of articles in *Journal of Object-Oriented Programming*,[1,2,3] we advocated a visual approach and discussed how we proceed in three stages using one particular visual language. In this paper, we will present a possible sequence of topics that follows the three-stage teaching methodology that can be used with ADA 9X or C++.

Topic Sequence

The following is one possible list of topics that can be used with ADA or C++ in teaching OOP with our proposed three-stage teaching methodology. For each topic, we provide a short description. These topics follow the three-stage teaching methodology. However, the point where one stage ends and the next stage begins is not clear cut.The point of transition depends on the types of students and the chosen language.

1. **Starting to Program**

   A very short program is presented. Although this is still in non-OOP stage, the first sample program will use an object. The reason we still call this stage non-OOP is because we do not actively push the concept of "object" to the students. They just use objects without really knowing about it. The types of objects we use in this stage are GUIobjects such as dialog boxes and Turtle objects (as in Papert's turtle graphics).

2. **Data**

   A simple description on variables, constants, types, and objects is provided. We basically want to address the data component of programs. The full definition of object is not yet provided.

3. **Control Flow**

   A language syntax and semantics of different control flow are explained. Here we start describing the behavioral aspect of objects. In other words, we explain how the control flow changes when we invoke an object's function.

4. **Characters, Array, Strings**

   In Topic 2, we describe a very simple data. Here we start describing a more complex data structure such as arrays and strings. Strings can be either the one that is a part of the chosen language or an object that is defined and provided by an instructor.

5. **Defining Objects - Part 1**

   This is the first place where students learn how to define their own objects. Since we are using a hybrid language, newly defined objects will not inherit from any existing objects. It is important to separate the notion of inheritance and defining an object. There is no choice in the pure object-oriented languages, but with hybrid languages, we can define an object without inheritance. By separating these two notions, students will learn the inheritance better, because students often erroneously equate object creation and inheritance mechanism (even with a hybrid language). Separating object creation and inheritance mechanism is very important especially for a language such as ADA 83, where object creation is allowed via its support of abstract data type. We also explain the different ways of passing parameters in conjunction to explaining how to define functions for an object.

6. **Defining Objects - Part 2**

   In this part 2 of object definition, we introduce and explain inheritance. Inheritance is nothing but one mechanism to speed up the creation of new object types. By proceeding in two parts, the students already know how to create an object type in the previous step. Now, they learn a mechanism to create a new type of object from the existing one. Only single inheritance is explained (even the chosen language supports multiple inheritance). Notice that the inheritance is introduced quite late in our proposed approach. In our opinion, too many instructors jump to the inheritance topic too soon. Inheritance is a very powerful feature that separates non-OOP and OOP languages. As such, we should take a good care to teach it properly by not introducing before the students are ready to fully understand it.

7. **Application Framework**

   The true power of object-oriented programming comes from the ability to use a large collection of pre-defined objects and maintain effectively in the class hierarchy (or lattice). To realize the potential of OOP, the students must familiarize themselves to the notion of looking at a huge collection of objects and intelligently choose a right object for a given task. We teach how to view and approach application framework (class library). As stated earlier, objects in the application framework are divided into four categories to facilitate easier management.

8. **Object-Oriented Design**

   The final topic is an introduction to a design methodology. Once the students have a solid foundation, we must provide a topic that will bridge the gap for students to the advanced concepts to be learned later in their studies. One of them is a design methodology for developing a large software system.

Summary

We described in the paper our proposed three–stage methodology for teaching OOP. Our teaching methodology can be adapted with any OOP language, and one possible list of topics is given here that can be used with ADA 9X or C++ (or any other hybrid languages).

References

1. Wu, C. T., "Teaching OOP to Beginners," *Journal of Object Oriented Programming*, Vol. 6, No. 1, March/April, 1994.

2. Wu, C. T., "Three-Stage, Incremental Approach in Teaching OOP," *Journal of Object Oriented Programming*, Vol. 6, No. 4, July/August, 1994.

3. Wu, C. T., "Teaching OOP in Three Stages Stage 3," *Journal of Object Oriented Programming*, Vol. 6, No. 5, September, 1994.

C. Thomas Wu is an Associate Professor of Computer Science at the Naval Postgraduate School in Monterey, California. Before joining the Naval Postgraduate School in 1985, he was an Assistant Professor of Computer Science at the Northwestern University in Evanston, Illinois. He received his Ph.D from University of California, San Diego. His current interests are database query languages, database design, object-oriented programming, and user interface.

Groupware User Experience: Ten Years of Lessons with GroupSystems

Jay F. Nunamaker, Jr.　**Robert O. Briggs**

Center for the Management of Information
University of Arizona, Tucson, AZ 85721

1.0 Introduction: A Team of Rugged Individualists

A great deal of organizational work gets done by individuals who set their jaws, put their shoulders to the wheel, and hustle. People are the real value in an organization. Technology only enhances the work they do. The first wave of computer technology focused on improving the productivity of individuals, and on automating routine tasks. However, many times people in organizations face problems that are not routine. They cannot be solved by the rugged individualist because no one person has all the experience, all the insight, all the information, or all the inspiration to accomplish a task alone. And so a team forms. Teams of people have accomplished the seemingly impossible, but teamwork brings its own set of problems. All who have suffered the grinding drudgery of meetings-without-end know how unproductive group work can be.

A relatively new breed of computer technology that is emerging targets the trouble spots for team productivity. It is known by many names: groupware, group support systems (GSS), computer supported cooperative work (CSCW), and electronic meeting systems (EMS), to name but a few. Besides supporting information access, groupware can radically change the dynamics of group interactions by improving communication, by structuring and focusing problem solving efforts, and by establishing and maintaining an alignment between personal and group goals. Organizations are now using groupware to leverage their existing information infrastructures in ways that have clearly improved overall productivity.

This paper will summarize the results of a decade of development and testing of electronic meeting systems at the University of Arizona and Ventana Corporation. Electronic meeting systems research began at the University of Arizona in the early 1980s. The result of that work was GroupSystems, an extensive electronic meeting system. In the late 1980s the university spun off Ventana Corporation to transfer GroupSystems technology to the private and public sectors. This paper will describe the technologies, discuss how and why they work, and present the lessons learned, both in the university laboratory and in the field, with thousands of users over a number of years.

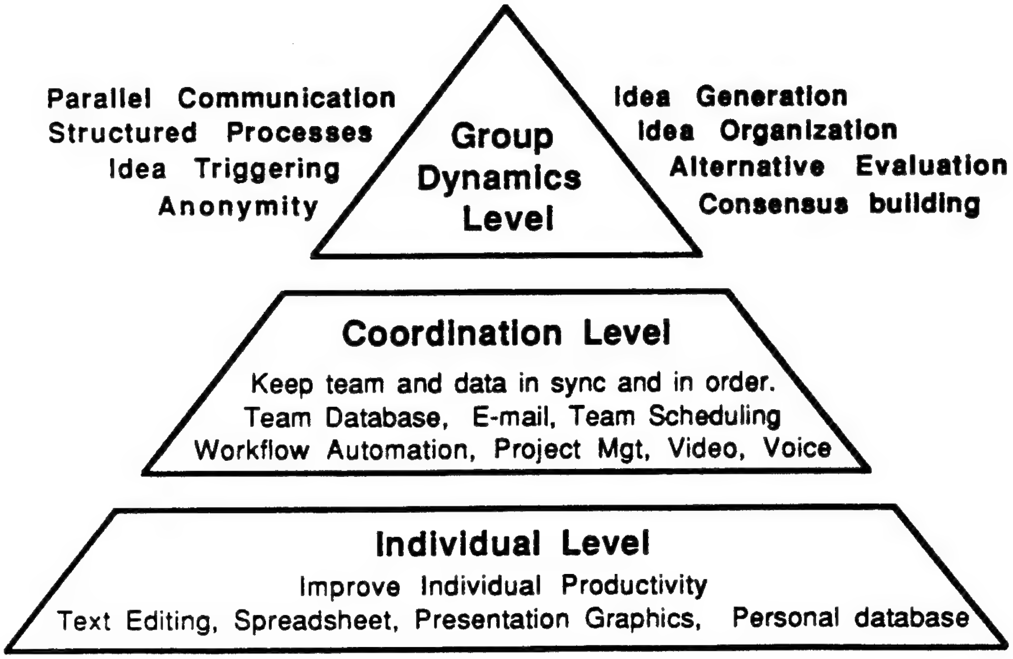
1.1 Three Levels of Technology Support for Teams

There are three levels of technological support for teams: the individual level, the coordination level, and the group dynamics level (Figure 1). At the most basic level, individuals in organizations work independently of one another. Each member uses stand-alone, individual-level tools like spreadsheets, word processors, or presentation graphics to become personally more productive. As the productivity of the individual rises, the productivity of the group as a whole may also rise. However, individual efforts may not be synchronized or coordinated, so individual productivity may not translate to organizational productivity.

In order to accomplish large complex tasks, the efforts of many individuals must be coordinated. At the coordination level, teams use network-based technology to assure that all the efforts of all team members are synchronized, and that all team resources are in order. E-mail, team scheduling, project management, team databases, and, of course, workflow automation are but a few examples of coordination technology. Lotus Notes is perhaps the best-known coordination-level application platform. Coordination support can greatly increase the productivity of organizations for problems that can be solved through a series of independently executed but related processes.

When a task requires that individuals with diverse skills and backgrounds jointly bring their expertise to bear on a problem, a team must form and the members must engage in collaborative work. Group-dynamics level technology addresses the special problems and opportunities that only arise from the dynamics of groups working together. Having now been used by more than a million people, GroupSystems is perhaps the most mature and widely used example of use of the technology at the group-dynamics level.

Figure 1.
Levels of Technological Support for Groups


Parallel Communication
Structured Processes
Idea Triggering
Anonymity
Group Dynamics Level
Idea Generation
Idea Organization
Alternative Evaluation
Consensus building
Coordination Level
Keep team and data in sync and in order.
Team Database, E-mail, Team Scheduling
Workflow Automation, Project Mgt, Video, Voice
Individual Level
Improve Individual Productivity
Text Editing, Spreadsheet, Presentation Graphics, Personal database

1.2 What is an Electronic Meeting System?

An electronic meeting system is typically based on a network of personal computers, usually one for each participant. EMS facilities often have one or more large public display screens (systems for geographically distributed participants often have software that substitutes for the public screen) (Nunamaker, et al., 1991).

1.2.1 Communication Support in an EMS

To understand the potential benefits of an EMS, imagine a meeting where dozens of other people will work on a pressing and perhaps politically touchy problem. Imagine that after the meeting leader has explained the problem, and on a pre-arranged signal, everyone started talking at once. Imagine further that in the cacophony everybody heard, everybody understood, and everybody remembered everything that was said by all the other participants. Imagine that all ideas were considered strictly on their merits, regardless of who offered them, and that as everybody gave honest and open opinions, nobody's feelings got hurt and nobody felt pressured or threatened by either peers or their boss. Clearly, this scenario would be impossible under normal circumstances, but an electronic meeting system can provide the communication support that makes it possible.

As the number of people in a conventional meeting increases, people spend more and more time waiting for a turn to speak. With little or no hope of getting the floors, some people withdraw, letting a few personalities dominate the discussion while good ideas go unspoken (Diehl & Stroebe, 1987). In an electronic meeting the opportunity to express an idea is never lost; everyone can "talk at once" by typing into his or her computer. The system makes all contributions available to the other participants almost immediately. This means that people do not lose track of their own ideas while listening to someone else, nor do they lose track of what others are saying while trying to remember what they want to say when they get the floor. All contributions become part of an electronic transcript. Strong (or loud) personalities can no longer dominate a meeting or sidetrack it into unproductive or irrelevant issues. All participants have an equal opportunity to contribute. Because their input can be anonymous, people can float unconventional or unpopular ideas without political risk.

Where conventional groups become less and less productive as the number of participants increases, electronic meeting systems permit dozens of people to work together. Productivity actually increases with group size. One reason for this is that large group size permits the inclusion of all key players in the same discussion. This reduces the number of times a meeting has to be interrupted or postponed until information or approval can be obtained from someone who is not present.

Large electronic groups also experience more instances of idea triggering. One person may say something that sparks an entirely new thought in someone else, a thought that may otherwise never have occurred. Furthermore, studies have shown that the quality of ideas generated is directly related to the number of ideas generated. Large groups supported by electronic meeting systems have been shown to generate more ideas of higher quality than groups of any size that do not use electronic meeting support.

1.2.2 EMS Support for The Problem Solving Process

An EMS is not a single piece of software, but rather a collection of computer-based tools, each of which can structure and focus the thinking of team members in some unique way. A system may include anywhere from 3 to 25 different tools, depending on its level of sophistication and the activities it is designed to support. For example, an electronic brainstorming tool encourages

Table 1
Summary of the Basic GroupSystems Tool Set

Electronic Brainstorming allows rapid generation of a free flow of ideas.

Idea Organizer gives structured methods for generating, synthesizing, and categorizing ideas

Vote helps evaluate ideas, measure consensus, and make choices using seven voting methods

Topic Commenter permits people to generate ideas and assign them to "file folders" or topics.

Alternative Evaluation compares a set of alternatives against a set of group-developed criteria

Policy Formation facilitates the process of developing consensus statements and action plans.

Stakeholder Identification helps the group to analyze the impact of actions or policies on identified stakeholders and on fundamental assumptions.

Questionnaire allows the group to respond to a questionnaire.

Group Matrix supports analysis of interrelationships between information sets.

Group Outliner allows a group to explore issues and develop action plans using a tree or outline structure.

Group Writer supports collaborative preparation of a single document.

Group Dictionary allows the group to develop common definitions for critical concepts.

Briefcase incorporates a set of resident utilities, including Calendar, File Reader, Notepad, Calculator, Clipboard and Quick Vote.

a group to diverge from standard patterns of thinking to generate as many unique ideas as possible. All users type their ideas into the system simultaneously. The system randomly passes ideas from one person to the next. The participants can argue with or expand on the ideas they see, or can be inspired to a completely different line of thinking. An idea organizer, on the other hand, encourages a group to converge quickly on key issues and explore them in depth. Electronic voting tools can uncover patterns of consensus, and focus group discussion on patterns of disagreement. Table 1 describes some of the most commonly used tools in the GroupSystems toolkit.

GroupSystems is a suite of tools to support the activities of a fundamental problem-solving process. The process is ubiquitous; it has been described in every discipline:

Notice the problem
Understand the problem
Develop alternatives
Select a solution
Plan for implementation
Act,
Monitor results.

Every phase of the problem-solving process requires information gathering, idea generation, idea organization, and idea evaluation. These activities are often accompanied by idea exploration and exposition.

The GroupSystems toolkit focuses participant efforts on the activities appropriate to the task at hand (Figure 2).

A group can use any of several tools to support a given activity, depending on how they wish to structure the group dynamics.
An electronic meeting system is more than just software for electronic brainstorming and electronic voting. An EMS includes not only the software, but also the processes and methods to accompany the use of the tools as well as the environment in which the tools are used. Handled with skill, an EMS can enhance group productivity dramatically. EMS is a new paradigm for collaborative work. The experience of doing business with an EMS is so different from conventional group work that many people have difficulty understanding why they would ever want to use it until they have experienced it personally. Having tried it, many people feel thwarted if they have to return to conventional groupwork methods.

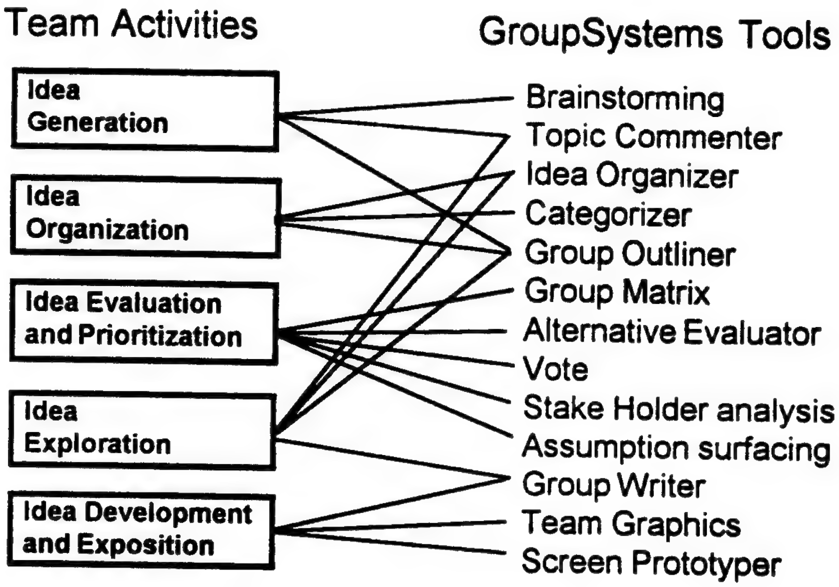
1.2.3 EMS Support for Information Access

An electronic meeting system can support the information access process in several ways. First, many group support tools include modules that permit users to define key words and phrases which the system will then use to scan and filter external information sources. Such tools can significantly reduce information overload.

Second, an EMS permits much larger meetings to be conducted productively, which means that a much larger information base is actually present in the minds of the meeting participants. In many cases this appears to be an important key to increasing productivity

Third, electronically supported interactions result in electronic transcripts. These transcripts themselves become a valuable information resource for the organization. For example, one aeronautical manufacturer had been using GroupSystems for nearly two years when a manager faced a particularly intractable problem with a vendor. He asked that the meeting transcripts be searched to find out how many times in the previous two years a particular problem had been mentioned along with the name of that vendor. There had never been a meeting about that particular problem, nor had there been a meeting about the vendor. However, the transcripts showed 19 different instances where the vendor and the problem were linked. Armed with that information, the manager was able to gain major concessions from the vendor and correct a problem that had been plaguing the production line for years.

Figure 2
Mapping GroupSystems to Problem-Solving Activities



1.2.4 EMS Support for Alignment of Goals

People always bring personal goals into a team effort. The team can only be productive to the degree that the goals of the team are congruent with the goals of the individuals on the team. Over the long term, people will not work against their perceived self-interest. There are a number of features and functions in an electronic meeting system that encourage the alignment of group and individual goals. Some tools permit anonymous input, which encourages people to speak up immediately if they perceive their ox is about to be gored. Because larger numbers of people can work together effectively, the whole group can learn about individual constraints early in the process, rather than having concerns fester under the surface, only to bring a project down after the investment of much time and expense. Some EMS tools specifically ask team members to identify the stakeholders in a project and to explicitly state their assumptions about each of them. This activity often surfaces misconceptions and unrealistic expectations, permitting a re-alignment of group and individual goals.

2.0 Lessons From the Field

Over the last decade we have learned a great deal about electronic support for team work: what features and functions are important, how to wield the tools effectively, and what kinds of results are possible. Perhaps the best place to begin a discussion of lessons learned is with the bottom-line results from EMS use, because without the payoffs the other lessons are of little interest.

2.1 How Effective are Face-to-face Electronic Meeting Systems?

Most of the early EMS work focused on improving the productivity of face-to-face collaboration because the dynamics of traditional group work were already well understood and the problems were only too clear. The first field trials of GroupSystems took place at the Owego, NY plant of IBM Corporation in 1986 (Vogel, et al, 1990). In a year-long study 30 groups used the technology to solve problems in production-line quality. Teams using the technology saved an average of 50% in labor costs over conventional methods. They also reduced the elapsed time from the beginning to the end of their projects by an average of 91%. The results were so dramatic that they were suspected of being anomalous, a fluke of the circumstances surrounding the study, so a second year-long study was conducted at six other IBM sites, each with a different set of business problems. In the second study, which tracked more than 50 groups, average labor costs were reduced by 55% and elapsed times for projects of all types were reduced an average of 90% (Grohowski, et al, 1990). IBM now has more than 90 electronic meeting rooms, and the number continues to grow.

In 1991 Boeing Corporation ran an independent study to determine whether there was a good business case for the use of electronic meeting systems. Over the course of a year they carefully tracked the results of 64 groups that were using the technology for problem definition, alternative generation and evaluation, and implementation planning. The result was an average labor saving of 71% and an average reduction of elapsed time of 91%. A conservative evaluation of the return on investment for the pilot project was 170% the first year (Post, 1992).

Figure 3
Bottom Line Benefits of GroupSystems in the Field

	Average Labor Saved/project	Flow Time Saved	Total Saved
IBM	55.2% $2,642	92%	$79,272
Boeing	72.0% &7,242	65%	$383,841

IBM:	30 projects	Mean 8.2 people	5.6 hrs/person
Boeing:	53 projects	Mean 10.5 people	7.5 hrs/person

Besides finding quantitative benefits, the IBM and Boeing studies documented improvements in the quality of results and the satisfaction levels of the participants. Since these studies, other organizations have conducted independent evaluations of the benefits of electronic meeting systems. The US Army reported a total savings of $1 million using GroupSystems to design a new Army-wide personnel tracking system. BellCore found a 66% reduction in labor costs for teams using the technology. (Three other case studies are discussed later in this paper.)

The field studies clearly show that substantial benefits, both economic and intangible, accrue from the use of electronic meeting systems. Experience also shows that success depends on both the way the tools are designed and how they are used.

2.2 EMS Lessons on Leaders and Leadership Style

Leader characteristics range from democratic to autocratic; situations, from chaotic to static; and organizational cultures, from fragmented to cohesive. The best leaders vary their styles according to situations. An electronic meeting system does not replace leadership, nor does it enforce a particular leadership style. Rather, it enhances a leader's ability to move a group forward in a given set of circumstances.

2.2.1 The Democracy Paradox

Organizational problem solving often requires collaboration among cross-functional teams of specialists. These teams, which often comprise experts of equal level, call for democratic leadership to coordinate communication, facilitate the group process, and make sure resources are available. A leader must be sensitive to the subtle cues that indicate whether a team is approaching consensus or spinning its wheels, but it is the team that establishes priorities, sets goals, and decides how best to advance them.

As any leader can tell you, however, democratic processes bog down decision making in endless meetings, conflicting proposals, and narrow interests. When crises arise in quick succession, a strong, autocratic leader is often needed to make rapid decisions. A centralized approach is not practical for most ad hoc teams, however, and therein lies the rub.

Electronic meeting systems make it possible to involve more people in arriving at decisions while ensuring that decisions are timely. Larger group size helps to keep the big picture in focus and eliminates a group leader's need to communicate separately with smaller subgroups. Because all subgroups can be represented at an electronic meeting, all perspectives may be heard without jeopardizing the speedy response necessary in a crisis. As the group builds an understanding of problems and tasks, there is less wheel spinning, more cooperation, less chaos, and more acceptance of decisions.

Traditionally, the larger the team, the slower the democratic decision-making process. More subgoals must be considered and less time is available for each person to speak. One solution is to break a large group into smaller ones with narrower responsibilities.

Table 2
Key Lessons for Outstanding Participation

- Anonymity: Otherwise key comments are not made.
- Participation: Parallel nature of the interaction increases participation. Participants sense they are part of the plan, that they are moving towards consensus and resolution.
- Group Size: Good ideas are a function of the quantity of ideas generated. More comments mean more good ideas.
- Evaluative Tone: When participants criticize ideas, performance improves. Keeps the group searching for better answers.
- Triggering: Any idea may inspire a completely new idea that would not have otherwise occurred.
- Peer Pressure: In face-to-face groups peer pressure keeps people moving. Distributed groups tend to lose momentum.
- Voting: Focuses key items for discussion, rapidly surfaces differences of opinion.

Subgroups can operate democratically, but their leaders must still resolve any intergroup problems that arise. It also becomes harder for individuals to see the big picture from the narrow viewpoint of the subgroup. Computerized group support systems increase the practical size of a democratic group from a handful to several dozen. Thirty individuals can share their ideas in the same amount of time that two or three could in a conventional meeting. Furthermore, electronic tools can organize the team's contributions in a form that can be assimilated faster and more easily than a sequential verbal stream.

It is also possible to use an EMS to permit experts from different geographic locations to participate in a discussion on a few minutes' notice. However, there is still much to be learned about how to manage distributed collaboration successfully. Presently it is far easier to receive advice from a distance than to complete full projects without ever working face-to-face.

Table 3
Key Lessons From Facilitators and Session Leaders

* Pre-planning is critical.

* Find a fast, clean way to do idea organization -- people hate it and you will lose them if you take too long.

* The group must always see where they are headed and how each activity advances them towards the goal.

* Divide work between face-to-face and asynchronous activities. Don't do anything in a meeting that can be done in an office.

* Expect that ideas generated will change the plan and the agenda.

* Mix modes between electronic interaction and verbal/oral interaction. Change locations and alternate between large and small groups every few hours to minimize burnout.

2.2.2 Re-aligning Leadership with Group Values

EMS technology can help resolve counterproductive conflicts between leadership styles. One man, who considered himself to be very democratic, presided over weekly 2 1/2 hour planning meetings with his staff. For the first hour and a half he would let the staff speak, but then he'd grab a felt marker and move to the whiteboard with the comment, "Let me see if I understand what you're saying." He would then describe his own agenda using words and phrases culled

from the group discussion. This vice president's superiors recognized the problem and decided to try using an EMS to alter his autocratic management style.
The staff was enthusiastic about the results, but he was not; he could no longer dictate the agenda, and he ultimately decided to stop using the system. The staff, with the support of top management, refused to let him. Group morale rose quickly, and the team prospered under a new, shared vision.

2.2.3 Leadership Pitfalls

Failure to make a meeting's objectives explicit can lead to disenchantment, particularly when participants spot phony democracy. If a leader includes a group in the decision-making process after the fact simply to "let them feel ownership," the group process breaks down. Leaders who merely want a team to understand a problem before they propose a solution should say so up front. If the objective is to develop a set of alternatives and recommendations, if should be so defined. Once the team has been commissioned to make a decision, however, a leader can contribute, advise, and argue, but the team will rebel against a leader who overrides its collective judgment.

Table 4
Key Lessons Regarding Social Behavior

* Technology does not replace leadership.

* Technology will support any leadership style.

* Some people resist EMS: The game has changed, oral/verbal skills and ramming an agenda through a meeting are <u>not</u> as important.

* Loss of engagement for distributed teams: The lack of visual and non-verbal cues appears to reduce the involvement of remote participants.

* Change of emotional engagement for face-to-face teams: The technology makes work more exciting for some, more ordinary and mundane for those who feel a loss of power.

* There is a need to develop group incentives.

* Be willing to accept criticism of you and the organization.

* Make sure there is an individual incentive to contribute to the group effort.

False promises of anonymity are equally damaging. Any attempt to find out who said what in an anonymous session undermines the leader's credibility and defeats the purpose of anonymous input, which is to solicit risky, unpopular, or opposing viewpoints. It is interesting to note that people often try to guess who said what in an anonymous session. Indeed, they are sometimes quite sure. Experience has shown, however, that such guesses are most often incorrect.

2.2.4 Role Clarification

Group support systems can also be used to identify those with a stake in a project, and reveal underlying assumptions. When a national library attempted to develop a computer system, it formed a team composed of representatives from different departments such as circulation, cataloging, acquisitions, and computing. For several meetings the groups tried and failed to develop a shared vision of the project. The team leader decided to use an electronic stake-holder and assumption-surfacing tool.

It turned out that the various departments had unrealistic expectations of the computer group, and the computer group had unrealistic expectations about the others. During the next few months, through vigorous and sometimes acrimonious debates, the team arrived at a common understanding and a shared vision. Until the participants engaged in stakeholder analysis they had not even been aware that fundamental differences existed. The group support system allowed them to share critical information and correct mistaken assumptions, solving an intractable problem and fixing a major oversight in the process.

Any tool is only as good as the artisan who wields it. This is just as true of sophisticated group decision support software as of a screwdriver. To realize these systems' enormous potential to expand the productivity of today's team-oriented organizations, leaders must recognize both tangible and intangible benefits.

The intangibles, which depend heavily on the style and quality of leadership, include greater group cohesiveness, better problem definition, a wider range of higher quality solutions, and stronger commitment to those solutions. The tangibles, already demonstrated, are dollar savings through greater productivity and reduced staff hours to reach decisions. On the bottom line, more time is free from the demands of frequent--and often frustrating--meetings.

2.3 Lessons about EMS Application Software

One of the key constraints on group productivity is that people can only pay attention to a few things at a time, and group work involves many competing demands for attention. Therefore everything in an EMS must work together to reduce distractions for a group. Over the years we have learned a number of lessons about what is important for successful EMS software in terms of structure, use, and interface.

2.3.1 The Values of Modularity

Interface Choices Affect Group Dynamics. It turns out to be very useful to build EMS software into a collection of special purpose modules rather than using a single module. For example, it is possible to build a single tool that can be used for idea generation, idea organization, idea evaluation (voting), and idea exploration. However, there are good reasons for separating these functions into different tools. First, any piece of software will impose constraints on how a group works, depending on how its features are implemented. An idea generation tool that only permits five lines of input per idea, for example, will encourage breadth rather than depth. An idea generation tool that permits unrestrained comment about a few items will encourage depth rather than breadth. Because interface choices will affect group dynamics, and because group dynamics are a critical concern for group productivity, it is useful to build separate modules, each to support a particular dynamic.

Table 5
Lessons About Groupware Application Software

* Interface choices affect group dynamics.

* Separate special purpose modules permit flexible process design.

* Templates assist mapping tools and process to group tasks.

* Group support must integrate with individual desktop application.

* Simplify the interface. No more than 30 seconds of instructions. It can never be too easy to use.

* Data must move from module to module seamlessly.

* Processes must be self-explanatory.

* System be robust in terms of stability and data recovery. An obvious point, groupware has many more ways to go wrong, and a higher failure cost than individual software.

* Users must have ready access to external data and past session transcripts.

Simple Interfaces are Vital. Another reason for building separate modules is that group interfaces must be kept very simple. Group members must talk, listen, think, and remember what has been said. Doing any of these things limits the ability to do the others well. If the computer interface poses an additional distraction it will hurt rather than help group productivity. By building separate modules for each activity it is possible to design interfaces that are so obvious that the user has no question about what the group is doing and how it is to be done. The screen has no extraneous cues. In the GroupSystems development effort we attempted to create tools that would permit groups to begin productive work with less than 30 seconds of instructions. Users are often able to begin work with no instructions at all.

Pre-Planning Is Crucial. The division of the EMS into modules to support a variety of dynamics and to simplify interfaces empowers a leader to move a group quickly and effectively toward a goal, but it also requires that the leader understand that different tools will have different effects on group behaviors. This means that the leader will have to plan ahead, deciding exactly what to get done, and then mapping a structured process using the appropriate tools to reach the goals with a minimum of effort.

2.3.2 Process Templates

Because many group problem-solving efforts will involve similarly structured efforts, it is very useful to build an EMS environment that can provide templates for sequences of activities. For example, many groups follow a brainstorm-organize-vote-explore-vote pattern. Another common sequence is generate solutions-generate criterion-evaluate solutions-select a solution. Having a set of standard templates for processes can make it easier for a group to decide what tools to use and what processes to follow. There are two forms a template can take.

First, it can supply a pattern for deciding which tools will be used in what order in conjunction with what group processes. Second, it can design a structure determining which features of a particular tool will be enabled during a given process. Both process-level and tool-level templates permit a leader to quickly adapt the EMS to team goals.

2.3.3 Data Portability

When building an EMS in modules it is also critical that the designer provide a simple and seamless way to move group information from one module to the next. For example, if a group spends time generating a broad set of ideas and then wants to evaluate which ideas are best, it must be easy to move the ideas to the voting module without undue effort. Long or awkward transitions between modules will disrupt the group dynamics and ultimately doom group processes to chaos.

Even when people are working as a group, there are still pieces of the work that will be done by individual members at their own desks. It is therefore desirable to integrate group productivity tools with individual productivity tools wherever possible. It is useful to be able to move information to and from spreadsheets, text editors, data bases, and other individual productivity applications.

2.3.4 Stability and Robust Recovery

It is an obvious point that all software must be stable and that data must be protected from loss. However, with groupware the attention to robustness must be carried to a near religious passion. Groupware by its very nature is more prone to failure than stand-alone software. Hundreds or thousands of events occur simultaneously and randomly in the system, and the system is made up of dozens of computers all acting in concert. Computers fail, networks fail, and people using them fail. An electronic meeting room full of busy and expensive senior executives will not tolerate systems that "hiccup". They will not be understanding when a morning's work disappears in a puff of virtual smoke. In a large group, many person-days can be invested in a few hours of electronic collaboration. When the system goes down the data must be safe. This means that all contributions should be written to disk almost as soon as they are contributed. Nothing should be maintained solely in volatile memory. Furthermore, data should be stored in more than one location, and the system should back itself up frequently.

2.4 Lessons About Anonymity and Evaluative Tone

An electronic meeting system can permit people to make anonymous contributions to group efforts. Putting staff members in a room full of networked workstations where they can express their true feelings may have about as much appeal for a project manager or CEO as joining a weekend encounter group. But the results of adopting the technology turn out to be shorter, more productive meetings and a freer flow of ideas. Because ideas enter the system and are circulated without attribution, EMS frees people to spark ideas off one another or to criticize ideas without fear of rebuke from peers or superiors. It encourages people to participate in meetings without inhibition and reduces the tendency for a few to dominate a meeting. A manager at Hughes Aircraft observed, "People who are usually reluctant to express themselves feel free to take part, and we've been surprised by the number of new ideas generated. We also reach conclusions far more rapidly."

2.4.1 Dealing With Criticism

When they first hear about anonymous input some people express concern that the discussion will quickly devolve into "flaming" sessions where participants launch vitriolic personal attacks laced with four-letter words and slanderous epithets. In tens of thousands of sessions in business and government organizations, however, we have not seen a single such disintegration.

This does not mean, however, that people are not critical in electronic meetings. They are. Participants will often raise issues that would never come out in face-to-face discussions. There is less sting in an anonymous electronic criticism than in a direct rebuke during a face-to-face meeting. The screen buffers the negative emotions that may accompany such criticism. Because nobody know where a particular idea came from people criticize the idea rather than the person who presented it. However, we have seen egos get bruised and people having difficulty dealing with honest feedback.

For instance, after asking for feedback on a reorganization plan, the president of one high-tech company was told that there were problems with the plan but that the staff could handle them.

In an EMS meeting, however, he found out what his management team really thought.

During the discussion, his staff responded both verbally, as in conventional meetings, and anonymously, by entering comments into their workstations. Everyone started talking and typing at once. A list of ideas scrolled down the large overhead screen behind the podium. Rapid-fire key clicks were an indicator of the energy that the meeting generated. People suggested options and argued over alternatives, inspiring one another to think in new and some times unexpected ways. But it was through the EMS that negative comments emerged: "The new plan doesn't stand a chance. It addresses the wrong issues entirely." "Once we've spent all the money to do this, the real problems will still exist, only worse." "We're way off-center with this one.".

After 40 minutes, the president was baffled. "We've been working on this plan for a year. Why didn't you people tell me this before? What do you think we should do now?"

Anonymity may also encourage group members to view their ideas more objectively and to see criticism as a signal to suggest other ideas. "I wasn't as uncomfortable when I saw someone being critical of someone else's idea, because I though 'Nobody's being embarrassed here at all,'" says Sam Eichenfield, president and CEO of Greyhound Financial.

"I noticed that if someone criticized an idea of mine, I didn't get emotional about it," says the Hughes Aircraft manager. "I guess when you are face-to-face and everyone hears the boss say, 'You are wrong,' it's a slap to you, not necessarily to the idea."

Despite the safe haven it provides for most participants, EMS isn't always so comfortable for the leader of a project or enterprise. Sometimes it takes courage for a manager to deal with the issues that surface in an anonymous meeting. It's hard to learn to deal with unpleasant input, but if problems lie buried for too long, they may become intractable.

In a rare incident, the founder of a very successful medical technology firm called together key personnel from multiple levels in the organization for an EMS session. Thirty minutes into the meeting he turned red in the face and stood up. Pounding a fist on his PC for emphasis, he shouted, "I want to know who put in the comment on the problem with the interface for the new system. We're not leaving this room until I know who made that statement?" He glared around the room waiting for a response. Everyone greeted his outburst with silence.

After a week's reflection he returned sheepishly to the group and said, "I had no idea there was trouble. I guess I'm more out of touch than I ought to be. Let's try again."

2.4.2 Diminishing Dysfunctional Politics

Anonymity helps to separate ideas from the politics behind them. Ideas can be weighted on their merits rather than on their source. Each member of a team tends to view problems from his or her own perspective, often to the detriment of the project or enterprise. For example, in traditional meetings engineers see engineering problems, sales people see marketing problems, and production people see manufacturing problems. In discussion and exchange of ideas anonymously from many different viewpoints, the big picture is more likely to emerge. EMS groups often achieve a unified, shared vision of problems and solutions--something that's difficult with traditional meeting methods.

EMS can translate negative comments into a positive influence on group productivity. Groups that are only allowed to make positive comments tend to stop looking for solutions when they have identified only a few. After all, everyone seems to like the ideas that have already been generated. On the other hand, when people are allowed to anonymously criticize anonymous ideas, people are not so sure they've found the best answer right off the bat. They continue to search for solutions until they have exhausted the possibilities. (Connoly, Jessup, & Valacich, 1990).

2.4.3 Extending the Search

Traditional groups, because they accentuate the positive, tend to cut meetings short, believing they've done a better job than they really have. Groupthink sets in when participants latch on to an idea too early and completely miss other, more promising options. The general attitude seems to be one of self-congratulation: Why go on if everyone likes the ideas we have already?

By the time the group has decided on a short list of options, everyone has had a chance to think through a position and has a better sense of how the group feels. At this point many people voluntarily abandon anonymity and begin to discuss and defend ideas openly and assign responsibility for action items.

While participants are less likely to be censured during an electronic meeting, they are equally unlikely to win much praise. Occasionally, new electronic meeting participants are concerned that they won't get credit for a really good idea. Establishing a group reward structure, rather than one based on individual performance, diminishes this concern. However, a really good idea still cries out for its author to speak up and take credit. Even in anonymous sessions, people sometimes sign an idea of which they are particularly proud.

2.5 Lessons on Electronic Voting

As corporations rely more on teams, with increasing emphasis on participative management, their need to create and measure consensus grows. In most cases electronic voting tools play a very different role from those of conventional voice or paper-ballot methods of voting. Traditional voting usually happens at the end of a discussion, to close and decide a matter once and for all. Electronic voting, however, tends to inspire a "vote early, vote often" approach. Because it is so fast, teams use electronic voting to measure consensus and focus subsequent discussion, rather than to close debate. In these ways, the technology is more accurately described as polling than as voting. While it can shorten discussions, saving time is not the only reason to use electronic polling tools. Teams find that polling clarifies communication, focuses discussion, reveals patterns of consensus, and stimulates thinking.

The following case studies, taken from confidential research of actual events, illustrate the diversity of benefits organizations can derive using electronic voting.

2.5.1 Confidence Voting

A management crisis loomed for a major telecommunications company. For six months, 39 senior managers had wrangled to come up with an ordered ranking of 89 technical researchers on the company's payroll. When they finally completed this arduous task, a new vice president rejected the process by which they had achieved their results.

This vice president didn't believe that the results accurately reflected the technical researchers' qualifications. An outside consultant was hired to engineer a new computer-supported voting process. The new scheme required each participant to submit both a ranking of each researcher and a measure of how strongly they felt about the ranking they were giving. The senior managers then reviewed several different graphical analyses of their votes and found much confidence and consensus on some of the rankings, and a great deal of variation on others.

Subsequent discussion revealed that many managers did not know some of the people they were ranking, relying instead on second-hand information and public opinion. After much discussion and information sharing the group voted again, this time with a much stronger consensus.

After the second vote the group discussed their remaining differences and in short order arrived at an overall ranking of their technical staff that all participants could live with. They agreed that the new computer-supported voting process was much more efficient than traditional voting methods and inspired a more open and focused exchange of ideas. More importantly, everyone from the vice president down felt that the new rankings were more legitimate than those obtained from the earlier process. The confidence-weighted votes and graphical representations of voting patterns provided managers with a larger picture than they had previously seen.

2.5.2 Getting Past Violent Agreement

Sometimes members of a team will vigorously debate issues upon which they actually agree. A startling example of this phenomenon of unneeded debate occurred in a health care organization that encompassed a dozen hospitals throughout a major metropolitan area. Three interest groups--doctors, administrators, and directors--set out to define a mission statement and to decide how various special services should be distributed among the hospitals. For reasons that were unclear, the process degenerated into an acrimonious battle--at which point someone noted that it had been three years since the groups had met without their attorneys being present.

The groups decided that electronic polling might be helpful in locating the source of the conflict, and decided to perform an experiment. Approximately 200 people attended a meeting where every participant was given a hand-held, radio-linked voting box. Using a large public screen, a facilitator displayed a number of policy statements such as, "When patients need emergency care it shall be given without reservation, without reservation, regardless of ability to pay."

Participants voted by agreeing or disagreeing with each statement as it was displayed.

Prior to the meeting, it was assumed throughout the health care organization that doctors, as a group, were responsible for obstructing agreement and thus progress. The prevailing wisdom was that hospital administrators and directors were the peacemakers in the group, and that a good deal of their energy went into persuading the physicians to be less intractable. This assumption was destroyed by the results. Analysis of the votes by subgroups revealed that, contrary to everyone's expectations, doctors and directors were in nearly perfect agreement on every issue. It was actually the staff administrators who were out of step, although for three years the administrators had been telling the directors that the doctors were causing problems.

2.5.4 Voting To Surface Information

Sometimes people don't think to share critical information until they are puzzling over the spread of electronic votes. Traditional methods of measuring consensus that do not reveal group thinking patterns can prove costly. The head of a mining company used a computerized voting system for the highly charged political task of allocating a budget across multiple corporate sites and projects. He asked a number of key executives for their opinions, but the results of the first poll were widely scattered. No one seemed to agree on budget priorities.

The president pressed his executives in order to understand why their voting patterns were so dissimilar, given that they all presumably had the good of the corporation in mind. finally, one vice president ventured, "None of us really knows what goes on at all these places. We can't really make an informed recommendation."

The president then arranged to have electronic comment cards included on the ballot, and advised group, "If you know about a project, type in what you know. If you don't know, read what the others have typed." Within half an hour, the group had exchanged a great deal of information about the various projects and sites, and the subsequent vote-and-discuss cycle resulted in high consensus on the budget allocation.

As the team left the room, one of the vice presidents pointed at an item on the bottom of the budget priority list, and commented ruefully, "We dumped $5 million dollars into that turkey last year." An eager champion had pushed the project, and when no one had information to dispute his arguments, the management council had simply taken a chance. Traditional consensus building had failed to uncover people's doubts, whereas electronic polling had revealed people's true feelings about the project.

2.5.5 No More Mr. Nice Guy

Electronic polling can sometimes facilitate decisions that are too painful to arrive at using traditional methods. A corporation with a particularly difficult budget crunch chose to use an electronic polling system to help decide how best to downsize. In many previous meetings, the possibility of eliminating a large but ineffective division was raised but was set aside for fear of offending the division's head, who was a very personable and effective lobbyist for his employees.

Although the division was generally unproductive, no one wanted to hurt the manager's feelings by pushing to have the division eliminated. Instead, using traditional voting methods, the group consensus indicated that across-the-board cuts should be implemented. Everyone would bleed a little, sacrificing some efficiency in the interests of harmony.

When the electronic votes were tallied, however, it was clear to all involved that the most sensible and most widely supported alternative was to eliminate the ineffective division. In doing so the organization did not have to make potentially crippling cuts to mission-critical functions, and at the same time it distributed responsibility for the decision among the participants.

2.5.6 Limits on Electronic Voting

Not all electronic voting sessions are successful. Occasionally, when all the votes are in, all the terms are defined, and all the hidden assumptions have surfaced, it turns out there are fundamental and irreconcilable disagreements between parties.

A savings and loan company faced a crisis that threatened its survival. During most of the discussion people were optimistic that they would reach a consensus and proceed accordingly. Rather than converging, however, group members views diverged as electronic voting proceeded. An analysis revealed that the group was, in fact, made up of several factions with mutually exclusive, deeply held positions. The session came to an end with an agreement to disagree. The only thing the participants knew was that in light of the bitter disagreements they had uncovered, the viability of the current management team, and thus the company, was at stake. On the bright side, the team was now focused on the difficult problem, rather than wasting time squabbling about minor disagreements.

In addition to making face-to-face meetings more productive, electronic voting will probably play a critical role in supporting geographically dispersed meetings. Remote meeting participants lack such nonverbal cues as shifting gazes, body positions, and gestures that let speakers sense it's time for a discussion to move on.

Although many teams save time and money with electronic voting, it would be a mistake to view that as the technology's main advantage. Some groups spend more time on their deliberations when using electronic voting than with traditional methods. Research has shown that groups using structured voting schemes and response analyses to clarify communication and focus discussion consistently reach higher-quality decisions than groups using traditional voting methods. Electronic tools that permit any participant to change his or her vote at any time and provide a real-time display of group voting patterns, establish a different dynamic by indicating shifts in consensus. New network-based voting schemes permit a group to begin interacting long before participants arrive in the meeting room, and to extend interaction after the face-to-face meeting is over.

2.6 Lessons about the EMS Facilities and Room Design

Besides good software and hardware, a group needs a good EMS facility to be successful. EMS facilities range from the basic to the sublimely sophisticated, from the inexpensive to the massively costly. An electronic meeting environment need not be expensive to be successful, but there are some fundamental design considerations that can make sure of successful use of the technology.

2.6.1 The Public Screen

Most EMS facilities include one or more public screens. The interactions between participants tend to alternate between on-line engagement via computer and oral engagement with face-to-face discussion. Availability of the public screen or screens is a way to give the group a common focal point for discussion, as well as a way to share public information.

Table 6
Lessons about Electronic Meeting Facilities

* Lighting is extremely important.

* Public screen is important for focusing group attention.

* Allow sufficient desktop space for spreading papers.

* Provide space for social interactions like eating and chatting.

* Take frequent breaks -- electronic work is intense.

* Minimize background noise.

* Be sure everyone can easily see everyone else in the room. Partially or fully recess monitors into tables if necessary.

* Provide back-up systems for servers, user workstations, etc.

2.6.2 Lighting is Critical

Lighting is an extremely important consideration. Florescent lighting, for example, tends to wash out the images on a projected public screen. Incandescent lights can be focused on the areas that need light while not impairing the public screen. It is also important to attend to how lights and windows may cause glare on computer screens. Some installations have had to be completely rewired because participants could not see the screens well enough to do their work.

Lighting is also a useful way to direct and focus participant attention during a session. For example, during on-line interactions, lighting in the room can be reduced to spots on the work-surface next to each participant. During general discussion lighting can be raised in the whole room. During presentations lighting can focus on the front and center location. If a large room

is being used by a small group the unused parts of the room can be darkened to give a more intimate feel to the area being used. Independent control of a variety of lighting fixtures enhances the ability of the leader to move the group forward toward its goal.

2.6.3 Lines of Sight and the Work Surface

Some consideration must be given to the configuration of the work surface that will be made available to the participants. First and foremost, the participants must be able to see their screen clearly, but they must also be able to see one another clearly. Some electronic meeting rooms have the CPUs sitting on desktops, and the monitors sitting on the CPUs. The result is a "design by Kilroy" effect. People must strain to see over and around the technology. In such a setting people tend not to engage in the proceedings; they lose interest and participation drops. Ideally the monitors can be partially recessed into the desktop so people have clear lines-of-site to one another. Some room designers have completely buried the monitors under a glass panel in the desktop, completely uncluttering the surface. This approach turns out to be a mixed blessing. It is tough to create a glare-free environment on the flat surface, even with so-called glare-free glass. It is also difficult to keep the monitor viewing area free of papers and clutter during the meeting. The partially-embedded monitor turns out to be a good compromise.

Along with space for the monitor, the work area must provide room for participants to spread out at least two full-sized legal sheets of paper. Participants often need to work from documents while interacting in an electronic meeting room.

2.6.4 Social Space

It is important to provide social space along with the work space in an electronic meeting environment. Because the electronic environment permits full interaction from all participants at all times, it tends to be an intense place to work. People need to take frequent breaks, perhaps every hour-and-a-half to two hours. The social space should allow for serving of snacks and drinks, and should have ample room for casual conversation. It turns out that a great deal of important activity happens during the breaks. Besides clarifying positions and informally negotiating agreements, people build a rapport with one another that simply cannot be achieved during the computer-supported interactions. Many leaders find that when people will be working together for several days at a time in an electronic meeting room, the work goes much more smoothly and quickly if the whole group dines at the same restaurant and stays at the same hotel.

2.6.5 Noise

Background noise in an electronic meeting room can be distracting and wearing during a long session. The CPUs for the workstations and the fans on the public screen projectors are prime culprits. Anything that can minimize such noise will improve the usability of the room. Placing CPUs in closed cabinets is one solution, although one must ensure sufficient ventilation to keep the machines from overheating.

One should avoid the temptation to place the CPUs in a separate room. Keyboards, mice, and monitors can all be wired for long distance connection to the CPU, and all the noise and heat is eliminated. However, as technology evolves it is often desirable to add peripheral devices for each workstation. Lightpens, digitizer, sound cards, and video cameras, for example, are all available for personal computers, and all can be useful in electronic meeting rooms. However, each requires a special adaptor card that must be placed in the computer, and that cannot be done if the computer is in the next room.

2.6.6 Redundancy

It is important to provide redundancy for every key component of the electronic meeting room. Back-up systems such as a second file server, multiple facilitator stations, and replacement components for user stations should be available. When key people are involved in a critical interaction they will not tolerate "hiccups." This is also an argument in favor of having at least two public screens. Should anything go down, there is something in reserve.

2.7 Lessons from the Facilitators and Session Leaders

The person who chairs an electronic meeting is the leader or facilitator. This person may be the group leader, another group member, or a separate, neutral individual who is not a group member. Using a non-member enables all group members to participate actively rather than having to lose one member to serve as the chair. A non-member can be a specialist in EMS and group work, but may lack the task and group knowledge of a regular member. The meeting leader/facilitator provides four functions. First this person provides technical support by initiating and terminating specific software tools and functions, and guiding the group through the technical aspects necessary to work on the task. This reduces the amount of training required of group members by removing one level of system complexity. In some cases technical support is provided by an additional technical facilitator or technographer.

Second, the meeting leader/facilitator chairs the meeting, maintains the agenda, and assesses the need for agenda changes. The leader may or may not take an active role in the meeting to improve group interaction by, for example, providing process structure in coordinating verbal discussions. This person also administers the group's knowledge. In an EMS designed without support for meeting leaders/facilitators, any group member may change or delete the group memory. When disagreements occur, members' competition for control can create a dysfunction. While this is manageable for small collaborative groups, it is much less so for larger groups with diverse membership, where competitive political motives and vested interests exist. With GroupSystems, members can view the group memory and add to it at their own workstations, but in general only the meeting leader/facilitator can modify and delete public information.

Third, the meeting leader/facilitator assists in agenda planning by working with the group and/or group leader to highlight the principal meeting objectives and develop an agenda to accomplish them. Specific GroupSystems tools are then mapped to each activity. Finally, in an on-going organizational setting where the meeting leaders/facilitators are not group members, the session leader provides organizational continuity by setting standards for use, developing training materials, maintaining the system, and acting as champion/sponsor, which is key to successful technology transfer. The roles of the meeting leader /facilitator may also change over time. For example, after a group has some experience using EMS, the need for technical support and agenda planning advice may decrease.

The most basic principle for successful use of electronic meeting systems is that the task must be very obvious to the group, and the activity in which its members are engaging must obviously advance them toward accomplishing that task. Where a conventional meeting may wander for three or four hours before people realize it is off track, with a computer-based meeting can resemble a train wreck in about ten minutes if it is not well planned. If the participants feel that the technology is engaging them in irrelevant activities they will quickly grow hostile and refuse to continue.

The importance of preplanning cannot be over-emphasized. Before an electronic session, the session leader must define exactly what concrete deliverables the group will create -- be it a problem statement, a list of possible solutions, a documented decision, a plan of action, or whatever. Defining a deliverable can in itself be a difficult task, but without it an electronic meeting is likely to founder. Having defined a deliverable, the meeting leader must then decide on a process for achieving the deliverable. This requires an awareness of the electronic tools and the different dynamics each can produce. Having mapped out a process for achieving the goal, the leader must also be sure that appropriate people are included in the meeting. Any group with a stake in the outcomes can and should be represented. With electronic meetings this is much more feasible than with conventional meetings, because electronic meetings can include may more people without hampering group productivity.

The work strain imposed by electronic interactions can be reduced if the plan permits alternating between electronic and oral interactions. It can also be useful to alternate multiple small group and a single large group sessions every few hours. Varying the work environment in these ways reduces monotony, which in turn improves productivity.

3.0 Three Case Studies

For all the discussion above about GroupSystems and how best to use it, it is not always easy to imagine how it can be applied to real business problems. Consider, therefore, the following GroupSystems case studies.

3.1 Dr. James Gantt: The Army Research Lab and Re-Inventing Government

Dr. James Gantt, director of the Army Research Lab in Atlanta, GA, has been actively involved in the development and implementation of large information systems for the U.S. Army. He was also one of the first to introduce the use of Electronic Meeting Systems to improve the productivity of government work teams.

Dr. Gantt arranged for the Army Information Systems Engineering Command at Fort Huachuca, Arizona to use GroupSystems to help people from the eight world-wide Army Major Commands define common information systems requirements. Originally requirements definition was done by bringing people together from all over the world for up to nine weeks. They would be trained in a process, and then would try to define data elements, flows, and critical success factors. At the end of the nine weeks they hoped to have a document that could be turned over to a software developer.

The first computer-assisted meeting of the Army project was successful from Gantt's perspective. "In a single session, we were probably saving conservatively about $50,000 worth of travel expenses and $75,000 to $100,000 on personnel cost avoidance," Gantt said. "The conclusion of the group was that we got four to six weeks of work done in three and a half days."

Gantt's group has been helping the technology evolve further by requiring portable platforms for electronic meetings and looking for computer-aided software engineering tools for prototyping within groupware applications.

In 1993 Vice President Albert Gore asked top government IT managers to compile suggestions for Gore's National Performance Review task force, part of President Clinton's effort to re-invent government. Hundreds of IT managers from every branch of government attended a conference held by the National Academy of Public Administration (NAPA) in Warrenton, VA. The managers broke into seven groups and each committed to deliver a report to the vice president. Dr. Gantt arranged for one group of 45 people to use GroupSystems to capture and organize their ideas. The group generated ideas and organized them into a rough draft of the document.

At the end of the conference the EMS-supported group was the only one to have even begun its report. The other six groups spent most of the time deciding how to manage the logistics of getting the report done, and some of the groups got as far as listing ideas that should be covered in their report. At the end of the conference each participant of the EMS-supported group walked away with a 3.5" disk containing the proceedings. The participants returned to their offices and fine-tuned a 25-page report that they sent to Gore's review team.

According to an article in Government Computer News, Larry Bernosky, who heads NAPA's Center for Information Management said that the group suggested many changes in government procurement. Participants also focused on developing an information architecture for the proposed National Information Infrastructure effort, identifying needs for data and interface standards. The group suggested creating an automated national clearinghouse of best practices for IT managers. Interagency working groups could then help agencies share ideas and information on common functions and services they perform. The group suggested that the administration could save money by spending seed money to reward people who take risks in developing new shared systems.

3.2 Ginny Wilkerson: The Chevron Pipeline

Ginny Wilkerson, an analyst of groupware end user computing tools at Chevron Information Technologies Corporation (CITC) decided to use the GroupSystems EMS to increase the effectiveness of their Quality Improvement teams and to aid their re-organization efforts. Based on internal successes, CTC also began using EMS to assist their clients with business process re-engineering. The Chevron Pipeline company established 14 standardization and improvement teams to look at their critical business processes. The first of those teams began analyzing procurement services.

The team began its mission using standard meeting and QI tools, but by the end of the first day using flipcharts and sticky pads members were exhausted. They were not looking forward to getting all their data (which covered a conference room wall) organized so that they could work on it more easily. Thinking that there must be a better way, one member remembered seeing a CITC demonstration of GroupSystems. The group was skeptical about the system, but were soon reassured by its ease of use. Ken Michel, Financial Transformation Coordinator of Chevron Pipeline, said, "We did it in half the time and in far more detail than we could have manually."

The Chevron Pipeline team found the EMS to be particularly cost effective when team members were scattered across the country, and travel was not feasible but participation was essential. The system was based in San Ramon, but the participants were linked by wide area and local area networks so that team members at remote sites could participate in the sessions via their Pcs.

As a result of the successful completion of the Chevron Pipeline Project, Ginny Wilkerson received a commendation for her creative use of technology during a complex procedure and policy analysis. The commendation states that the net result of the effort will be a savings of over $5 million per year, and that the process would have been nearly impossible using traditional flip-chart methods.

3.3 Frank Lancione: The Defense Transportation System

Mr. Frank Lancione of Coopers & Lybrand was asked to help the Department of Defense with a major business process re-engineering project. The department of defense had decided to change the way they manage the transportation of people and supplies in times of crisis. A team of experts from every branch of the military assembled to tackle the job. They were charged to reduce costs, speed up response times, standardize transportation management across all services, and eliminate redundancy.

The group began working with traditional flip-chart meeting and modeling methods. The first workshop was so slow that the people involved feared the task might have been unachievable. The team agreed to try GroupSystems for their next gathering. The team reported that they were able to collapse three months work into two weeks. Subsequent workshops were all held using GroupSystems.

The time savings from the use of the electronic meeting system resulted in substantial cost savings, but it had a second benefit that was even more important. Because the workshops could be executed in two weeks, the top experts and managers from each department were able to participate. Under the old methods these people had not been able to attend because they could not been able to leave mission-critical jobs for three months at a time.

The workshop members began by modelling the military transportation process down to the transaction level. At first they modeled sealift, airlift, and land transportation as separate processes. However, when they examined the processes for transporting people and supplies first by land, then by air, then by land again, they discovered serious bottlenecks in the processes as the mode of transportation changed. The team therefore developed a unique three-dimensional planning process that included both intra-agency and inter-agency components over a long period of time. The EMS permitted the group to fulfill its mandate. The new system design provided reduced cost, improved response times, and an information system that would be much less vulnerable to disruption in the event of a sudden emergency.

4.0 Lessons Yet to be Learned

Although a great deal is known about how and why electronic meeting systems work, there are still many lessons still to be learned. For instance, to what degree can the role of the facilitator be automated and supported? Research is underway to apply neural network and other AI technologies to the organization of randomly generated ideas (Chen & Lynch, 1992). Other researchers are looking into the use of expert systems for planning group processes, matching tools to group needs, and monitoring group progress toward a goal. However, real-time use of expert systems for these purposes is still some distance away.

A majority of GroupSystems users walk away from electronic meetings feeling enriched and empowered by the technology and by the collaboration they've had with their colleagues. They often feel somewhat exhilarated by having covered so much work in so little time. Not everyone walks away feeling satisfied, however. A minority feel a loss of power, and find electronic meeting work mundane and ordinary. For instance, in a recent meeting of senior executives at a large high-tech manufacturing firm the participants conducted an annual tactical planning session. For each of the previous five years the session had taken three days. Using GroupSystems the executives achieved their goal in under two hours, and all agreed that the resulting plan was of better quality and had more detail than any plan they had ever generated. Still, when asked how they had liked the session the executives expressed a general sense of mild discomfort. "It moved pretty slowly," one said.

"It felt kind of routine," another replied.

"I've made my career on being able to drive an agenda through in a tough meeting," another added. "Today there wasn't any need for what I do best." Electronic meeting support is intentionally designed to eliminate many aspects of traditional meetings that tend to get the adrenalin pumping. Criticism stings less, you can't be interrupted, you can argue your points without pounding the table for emphasis. The result appears to be less emotional engagement for some, which can may mean less satisfying work. Researchers are only beginning to investigate the importance of this emotional engagement, what the consequences of its loss may be, and what can be done to restore it without reducing the benefits of the EMS.

Another problem is how to fully engage people in team work when they are geographically distributed. Present technology permits full participation by remote users, but experience shows that these users are much more easily distracted from the work than are people who are co-located. Phones ring, people knock on the door with papers to be signed, the user decides to check e-mail. Often the remote user winds up more an observer of the electronic meeting than a full participant. In the face-to-face meeting room peer pressure helps to keep people active. We often see somebody nudge the person next to them in a kidding fashion with an oral comment like, "Say, I don't hear any keystrokes coming from you. Did you go to sleep?"

Part of the answer for more complete engagement may come from a set of software features that replace some of the missing non-verbal cues that engage a person in a meeting. Adding video teleconferencing may help. An electronic public screen that is visible on the console of the remote user may also help. If all users were given telecursors that let them electronically point at objects on the public screen, this could substitute for missing hand gestures. Temporary electronic scribbling over the public screen with a pen-based interface may also assist remote participation. There is even the possibility that virtual reality may play a part in creating the illusion of shared presence for remote meeting participants, thereby engaging them more fully in the group activities.

The collision of telecommunication, cable, and computing may have a dramatic effect on electronically supported groups. Currently the cost of data communication channels is a major constraint on remote collaborative computing. A channel with sufficient bandwidth to support full-motion high-resolution video teleconferencing can cost as much as $10,000 per month. A nation-wide digital fiber network may reduce such a channel cost to less than one percent of that amount. With the ability to pass vast quantities of data to and fro, the meeting environment of the future may look quite different from the meeting environment of today.

5.0 The Meeting Environment of the Future

Technology exists today to permit team members to work any time, any place, and nearly any way they want. Electronically supported groups can work as individuals linked by electronic media in a virtual meeting; they can be located in several meeting rooms that are connected electronically, they can work at the same time, or different members can work at different times. They can work in small groups of four or five, or they can work in very large groups of 50 or more people. However, today many of these technologies are developing independently of one another. To date there is no single environment that combines individual and group support, remote and face-to-face collaboration, text, graphics, video, and voice links, and shared computer applications. Each working mode requires electronic tools specifically tailored to the situation. Each kind of collaboration requires different kinds of information. For example, some information is formal, as with corporate reports. Other information is less formal, as with hallway conversations or notes on the back of a placemat. As we move toward the future, all group support technologies will be integrated into the same working environment.

Three new experimental environments are now under construction at the University of Arizona: The Multimedia Collaboration Center, The TeamRoom 2000, and the Mirror Project. Research in these facilities will focus on the integration of many technologies to support group work, including automating traditional group processes, real-time video and audio through the computer, and intelligent information retrieval.

The Multi-media Collaboration Center will combine audio and video teleconferencing technology with GroupSystems and other electronic meeting support. It will support 29 users with workstations and three 10-foot public display screens. It will also provide a full arsenal of multi-media display equipment and will give users on-line access to an electronic library and other external databases. This center will also be fully integrated with the two other environments, the TeamRoom 2000 and the Mirror Project.

TeamRoom 2000 will extend the capabilities of existing group support facilities to create an any-time/any-place meeting environment. TeamRoom 2000 will combine pen, voice, and wireless network technology so that distributed groups can work in a virtual team space. A dedicated facility with a network of pen- and voice-based notepad computers will be the hub of this team environment, but users will move in and out with their portable computers as their needs dictate. Team 2000 will extend the Multimedia Collaboration Center concept beyond the four walls.
The Mirror Project will address questions relating to group-to-group interactions. Current audio and video teleconferencing technology does not support the rich and subtle communication possible in face-to-face meetings. Where the TeamRoom 2000 project will focus on the interface

between the individual and the group, the Mirror facility will focus on creating the illusion of presence that is lost when groups use today's video teleconferencing facilities. The Mirror facility will feature floor-to-ceiling high-resolution video display walls and will address optical, mechanical, and human issues to enhance the connection between remotely collaborating groups.

6.0 Conclusions

We have learned a great deal about how to engage in successful electronically supported collaboration, and we still have a great deal to learn. The field is growing rapidly in many different directions. Looking back ten years from now today's technology may look horse-and-buggy. Nonetheless, the fundamental principals of collaboration technology will still apply. There will still be problems so intractable that no single person will be able to solve them alone. Technology will still improve communication, still structure and support thinking processes, and still provide access to information. And technology will still be no substitute for leadership and a solid understanding of group dynamics. These principals will remain as constants.

References

Chen, H., & Lynch, K. "Automatic Construction of Networks of Concepts Characterizing Document Databases. IEEE Transactions on Systems, Man, and Cybernetics. 22, 5, 1992, 885-902.

Connoley, T., Jessup, L. M., and Valacich, J. S. "Effects of Anonymity and Evaluative Tone on Idea Generation in Computer-Mediated Groups. Management Science, 36, 6, 1990, 689-703.

Dallvalle, T., Esposito, A., & Lang, S. "Groupware - One Experience." The Fifty Conference on Corporate Communication: Communication in Uncertain Times Fairleigh Dickinson University, May 20, 1992, 2-9.

Diehl, M., and Stroebe, W. Productivity Los in Brainstorming Groups: Toward the solution of a riddle. J. Personality and Social Psychology, 53, 3, 1987, 497-509.

Grohowski, R. B., McGoff, C., Vogel, D. R., Martz, W. B., & Nunamaker Jr., J. F. "Implementation of electronic meeting systems at IBM." MIS Quarterly, 14, 4, 1990, 369-383.

Nunamaker Jr., J. F., Dennis, A. R., Valacich., J. S., Vogel, D. R., & George, J. F. " Electronic Meetings to Support Group Work." Communications of the ACM, 34, 7, 1991, 40-61.

Post, B. Q. "Building the Business Case for Group Support Technology. Proceedings of the 25th annual Hawaii International Conference on Systems Science, IEEE, 1992, Vol IV, 34-45.

Vogel, D. R., Martz, W. B., Nunamaker Jr., J. F., Grohowski, R. B., & McGoff, C. "Electronic meeting system experience at IBM. J. of MIS, 6, 3 (1990).

Dual Use Panel

Moderator: David R. Basel
Deputy Director, Ada Joint Program Office
Center for Information Management

Panelists: Joanne P. Arnette
Director, Software Systems Engineering
Center for Information Management

Ralph Crafts
Ada Software Alliance Incorporated

Deborah P. Dowling
Electronic Data Systems, Incorporated

Dr. Frances L. Van Scoy
West Virginia University

Ada DUAL-USE PANEL

Ada DUAL-USE WORKSHOP

The Ada Dual-Use Workshop was held in October 1993. During the workshop, over 175 representatives from the academic, industry, vendor, and government communities provided input and recommendations on ways to promote the adoption and use of Ada outside of the DoD. During the workshop panels were convened for each of the four communities.

Academic Panel Recommendations

1. Restore/initiate consistency in DoD's treatment of Ada and Software Engineering.

2. Expand an existing forum, ASEET (Ada Software Engineering Education and Training), to serve the needs for timely information and technology exchange among all educators that are stakeholders in Ada 9X.

3. Provide low cost/free environments, tools and other resources supporting instruction and research in and with Ada 9X should be made available to educators and students as soon as possible.

Industry Panel Recommendations

1. Strengthen the Ada mandate by enforcing the waiver process and adhering to public law.

2. Prepare business case for the use of Ada In order to place a business focus on Ada plans and strategies.

3. Develop bindings and COTS tools on a cost-sharing basis (Government could take limited rights; contractor could get marketing rights) in cooperation with academia, industry, and others.

4. Define a marketing strategy for the commercialization of Ada to include defining the market and market needs for Ada within the DoD.

5. Stimulate educational use of Ada by permitting language subsets and supersets and by providing grants and other forms of assistance to high schools, universities, and colleges for Ada-based software engineering training.

6. Provide reusable Ada libraries for all environments, domains, and platforms.

7. Provide incentives that will motivate investors to renew their Ada initiatives, and should stimulate industry to develop and recognize core competency in Ada.

Vendor Panel Recommendations

1. Demonstrate a consistent commitment to use Ada for DoD applications.

2. Partner with industry in promoting an 'awareness of Ada's benefits.

3. Create a business environment which is conducive to private investment.

4. Accelerate the availability of high quality Ada 9X products.

Government Panel Recommendations

1. Provide examples that are considered major Ada success stories.

2. Develop videos in conjunction with other federal agencies that identify existing programs, provide statistics, teach lessons that have been learned.

3. Clarify and promulgate the Ada mandate for the defense industry.

4. Focus on Ada bindings through identifying required bindings, working with industry and NIST to develop and publish standard binding specifications.

5. Support Ada tools for new language versions such as Ada 9X, just like "C" does.

6. Provide incentives to recognize Program Managers of Ada intensive projects.

7. Provide reusable Ada libraries for all environments, domains, and platforms.

8. Consider allowing non-validated Ada 9X compilers for non-critical systems until validated ones reach the market.

9. Subsidize more research in Ada and especially Ada 9X.

Ada DUAL-USE STRATEGY

The uppermost goal of the Ada Dual-Use Strategy is to reduce the cost of software for DoD systems (mission-critical, information, etc.) by exploiting the market potential of Ada. By creating both market push and pull, the DoD can stimulate outside investment, thereby reducing its overall cost of ownership. In order to accomplish this goal, the DoD is focusing on increasing the commercial use of Ada and the software engineering principles that it embodies. This commercial focus is where the true market potential for Ada resides and will allow the DoD to take advantage of Ada's dual-use potential.

During the workshop, DISA and the panels heard a wide variety of suggestions on how the DoD could meet it commercialization goals. After reviewing the panel summaries and recommendations, it became apparent that action was needed on the following five fronts:

improving the marketing of Ada; establishing partnerships with academic, industry, and government organizations; providing support and incentives to encourage the development of Ada tools; re-enforcing DoD's internal commitment to Ada; and maintaining support for the current activities of the AJPO.

(1) Ada's image must change via an aggressive publicity campaign. Although Ada has shown merits, Ada has received extremely negative press reports. Ada must be viewed as a success to be successful. Promotion and aggressive marketing are the primary keys to a successful image building campaign.

(2) Partnerships need to be established with elements from the academic, industry, and government sectors. These partnerships would encompass jointly funded efforts that provide a benefit and return on investment for both the DoD and its partners. "Win - Win" is the underlying theme of these partnerships.

(3) Technical support and incentives should be provided to help increase the use of Ada within the commercial sector. A market pull is needed to broaden the base of support. Bindings, tools, and educational initiatives need to be mounted to created this pull.

(4) Guidance needs to be provided for Program Executive Officers (PEOs)/Program Managers (PMs) to constantly interpret the Ada Mandate.

(5) The DoD needs to continue support for current AJPO activities including Core Ada, the Ada Technology Insertion Program (ATIP), and the Ada 9X Transition Programs.

By addressing these five major areas, the DoD will be taking a giant step forward in the commercial use of Ada in dual-use programs. Dual-use will only happen with teamwork, leadership, and cooperation both inside and outside of the DoD. The DoD needs to build partnerships with academia, industry, and the vendors to make the dual-use of Ada successful. The recommendations received from this workshop have incorporated into the Draft DoD Plan and DISA/JIEO/CIM is moving forward on making those recommendations a reality.

Ada9X Panel

Moderator: Christine Anderson, US Air Force

Panelists: Tucker Taft, Intermetrics
Invited presentations on experiences of Ada9X

1 Evolution of Ada 9X

Ada is a modern programming language suitable for those application areas which benefit from the discipline of organized development, that is, *Software Engineering*; it is a general purpose language with special applicability to real-time and embedded systems. Ada was originally developed by an international design team in response to requirements issued by the United States Department of Defense [DoD 78].

Ada 9X is a revised version of Ada updating the 1983 ANSI Ada standard [ANSI 83] and the equivalent 1987 ISO standard [ISO 87] in accordance with ANSI and ISO procedures. This present document describes the overall Rationale for the revision.

1.1 The Revision Process

Although Ada was originally designed to provide a single flexible yet portable language for real-time embedded systems to meet the needs of the US DoD, its domain of application has expanded to include many other areas, such as large-scale information systems, distributed systems, scientific computation, and systems programming. Furthermore, its user base has expanded to include all major defense agencies of the Western world, the whole of the aerospace community and increasingly many areas in civil and private sectors such as telecommunications, process control and monitoring systems. Indeed, the expansion in the civil sector is such that civil applications now generate the dominant revenues of many vendors.

After some years of use and many implementations, a decision was made in 1988 to undertake a revision to Ada, leading to an updated ANSI/ISO standard. It is normal practice to automatically reconsider such standards every 5 or 10 years and to determine whether they should be abandoned, reaffirmed as they are, or updated. In the case of Ada, an update was deemed appropriate.

To understand the process it should be explained that ANSI, as the US national standards body, originally proposed that Ada become an ISO standard. It is normal practice that ANSI, as the originating body, should sponsor a revised standard as necessary. The US DoD acted as the agent of ANSI in managing the development of the revised standard. Within the US DoD, the Ada Joint Program Office (AJPO) is responsible for Ada and the Ada Board is a federal advisory committee which advises the AJPO.

The revision effort began in January 1988 when the Ada Board was asked to prepare a recommendation to the AJPO on the most appropriate standardization process to use in developing a revised Ada standard, known in the interim as Ada 9X. The recommendation [DoD 88] was delivered in September 1988 to Virginia Castor, the then Director of the Ada Joint Program Office, who subsequently established the Ada 9X Project for conducting the revision of the Ada Standard. Christine M Anderson was appointed Project Manager in October 1988. Close consultation with ISO was important to ensure that the needs of the whole world-wide Ada community (and not just the defense community) were taken into account and to ensure the timely adoption by ISO of the new standard. Accordingly a Memorandum of Understanding was established between the US DoD and ISO [ISO 90].

The Ada 9X program consists of three main phases: the determination of the Requirements for the revised language; the actual Development of the definition of the revised language; and the Transition into use from Ada 83 to Ada 9X.

The output of the Requirements Definition phase was the Ada 9X Requirements document [DoD 90], which specified the revision needs which had to be addressed. The Mapping/Revision phase defined the changes to the standard to meet those requirements; it achieved this in practice of course by defining the new standard.

The scope of the changes was guided by the overall objective of the Ada 9X effort [DoD 89a]

> *Revise ANSI/MIL-STD-1815A-1983 to reflect current essential requirements with minimum negative impact and maximum positive impact to the Ada community.*

It is probably fair to observe that the changes which were deemed necessary are somewhat greater than might have been foreseen when the project first started in 1988. However, technology and man's understanding do not stand still and the changes now incorporated are very appropriate to the foreseen needs of a modern language for Software Engineering into the 21st Century.

1.2 The Requirements

The development of the requirements was a somewhat iterative process with a number of sub-phases. First, the community of users was invited to submit Revision Requests, secondly, these were sorted and analyzed in order to determine what was really implied by the requests, and finally, a coherent set of requirements was written.

Establishing the level of the requirements needed care. Quite naturally an individual user would often perceive a need in a manner which reflected a symptom of an underlying problem rather than the problem itself. It was very important that the requirements reflected the essence of the problem rather than what might in effect be simply one user's suggested solution to a part of a problem. One reason for this concern was to ensure that better, perhaps deeper and simpler, changes were not excluded by shallow requirements.

In some cases a complete understanding of a requirement could not be established and this led to the introduction of Study Topics. As its name implies a Study Topic described an area where something ought to be done but for some reason the feasibility of a solution was in doubt; perhaps the needs were changing rapidly or there were conflicting needs or implementing technology was unstable or it was simply that we had an incomplete understanding of some basic principles.

The general goal of the revision was thus to satisfy all of the Requirements and to satisfy as many and as much of the Study Topics as possible. Any unsatisfied Study Topics would be set aside for further consideration for Ada0X (that is the next revision due around 2003).

The requirements document described 41 Requirements and 22 Study Topics. These were divided into a number of chapters which themselves form two broad groups. The first group covered topics of widespread applicability, whereas the second group addressed more specialized topics.

The first group consisted of the need to support International Character Sets (originally identified by several nations in the 1987 ISO standardization process), support for Programming Paradigms (Object Orientation was included in this category), Real-Time requirements, requirements for System Programming (such as Unsigned Integers), and General requirements. This last topic included specific matters such as error detection and general considerations of efficiency, simplicity and consistency; examples of a number of individually minor but irritating aspects of Ada 83 which fall into this category were given in an appendix.

The second, specialized, group consisted of requirements for Parallel Processing, Distributed Processing, Safety-Critical Applications, Information Systems, and Scientific and Mathematical Applications.

The breadth of these specialized requirements raised the specter of a language embracing so many application areas that it would become too costly to implement in its entirety for every architecture. On the other hand, one of the strengths of Ada is its uniformity and the last thing desired was a plethora of uncontrolled subsets. Accordingly, the Requirements document recommended the concept of a Core language plus a small number of specialized Annexes as a sensible way forward (with emphasis on keeping the number of such annexes very small). All validated compilers would have to implement the Core language and vendors could choose to implement zero, one or more annexes according to the needs of their selected market places.

The Requirements document also included general guidance on the approach to be taken to avoid unnecessary disruption to the Ada community. This covered vital matters such as the ease of implementation, object code efficiency, keeping a balance between change and need, and upward compatibility. It finally stressed that support for reliability, safety and long-term maintainability should continue to take precedence over short-term coding convenience.

Of all these requirements there was strong emphasis on the overriding goal of upward compatibility in order to preserve existing investment by the Ada community as a whole.

1.3 The Main User Needs

The Requirements reflected a number of underlying User Needs which had become apparent over the years since Ada was first defined. Apart from a small number of very specific matters such as the character set issue, the specialized areas and generally tidying up minor inconsistencies, four main areas stood out as needing attention:

- Interfacing. Ada 83 recognizes the importance of being able to interface to external systems by the provision of features such as representation clauses and pragmas. Nevertheless, it was sometimes not easy to interface certain Ada 83 programs to other language systems. A general need was felt for a more flexible approach allowing, for instance, the secure manipulation of references and the passing of procedures as parameters. An example arises when interfacing into GUI's where it is often necessary to pass a procedure as a parameter for call-back.

- Programming by Extension. This is closely allied to reusability. Although Ada's package and generic capability are an excellent foundation, nevertheless experience with the OO paradigm in other languages had shown the advantages of being able to extend a program without any modification to existing proven components. Not only does this save disturbing existing software thus eliminating the risk of introducing errors but it also reduces recompilation costs.

- Program Libraries. The Ada program library brings important benefits by extending the strong typing across the boundaries between separately compiled units. However, the flat nature of the Ada 83 library gave problems of visibility control; for example it prevented two library packages from sharing a full view of a private type. A common consequence of the flat structure was that packages become large and monolithic. This hindered understanding and increased the cost of recompilation. A more flexible and hierarchical structure was necessary.

- Tasking. The Ada rendezvous paradigm is a useful model for the abstract description of many tasking problems. But experience had shown that a more static monitor-like approach was also desirable for common shared-data access applications. Furthermore, the Ada priority model needed revision in order to enable users to take advantage of the greater understanding of scheduling theory which had emerged since Ada 83 was defined.

The main changes incorporated into Ada 9X reflect the response to the Requirements in meeting these four key needs while at the same time meeting an overriding need for upward compatibility in order to minimize the effort and cost of transition.

1.4 The Approach

In responding to the revision requirements, the Revision team followed the inspiration of Jean Ichbiah (who led the original design team), both in remaining faithful to the principles underlying the original Ada design, and in the approach to the revision process. To quote Dr Ichbiah in his introduction to John Barnes' textbook on Ada [Barnes 82]

> *Clearly, further progress can only come by a reappraisal of implicit assumptions underlying certain compromises. Here is the major contradiction in any design work. On the one hand, one can only reach an harmonious integration of several features by immersing oneself into the logic of the existing parts; it is only in this way that one can achieve a perfect combination. On the other hand, this perception of perfection, and the implied acceptance of certain unconscious assumptions, will prevent further progress.*

Wherever possible, enhanced functionality in Ada 9X has been achieved by seeking and eliminating such unnecessary assumptions, thereby permitting the generalization of features already in Ada, and the removal of special cases and restrictions. A careful analysis was made of Ada 83, of language study notes prepared during the original design process and of the many Ada Issues and other comments that have since been received. Based on this analysis, and on the Ada community's experience in implementing and using Ada during the past ten years, the team identified limitations that, while they were included to simplify implementations and/or to lower risk when the language was first standardized, are no longer necessary. They also drew upon the wealth of practical experience gained during the 1980's in the use of object-oriented design methods, object-oriented programming languages, and real-time programming made possible by Ada.

The resulting Ada 9X revision is upwardly compatible for virtually all existing Ada 83 applications. Most incompatibilities are restricted to pathological combinations of features that are rarely used in practice. Total upward compatibility would not have allowed the correction of certain errors and certainly would not have allowed the enhancements needed to satisfy many of the requirements. Indeed, as discussed in the Appendix in more detail, no language revision has ever achieved total upward compatibility. The careful attention to this issue in the design of Ada 9X means that the expected transition costs for existing Ada programs are anticipated to be very low indeed.

Following the guidance of the Requirements document, Ada 9X comprises a Core language, which must be implemented in its entirety, plus several Annexes which provide extended features for specific application areas. These Annexes provide standard definitions for application-specific packages, pragmas, attributes, and capacity and performance characteristics of implementations. The Annexes address the following areas: Systems Programming, Real-Time Systems, Distributed Systems, Information Systems, Numerics, Safety and Security, and Language Interfaces. It should be noted that much of the annex functionality is already provided by implementations of Ada 83 but in non-standard ways; the Annexes will thus increase portability between implementations.

Environment Assessment Methodology Panel

Moderator: Richard Petersen, USN, Information Processing Directorate, DISA Center for Standards

Panelists:
Dr. Robert Balzer, Univ. of Southern California
Herman Fischer, Mark V
Herbert Kopplinger, Seimens Nixdorf
Dr. Charles McKay, Univ. of Houston-Clearlake
Myer Morron, Independent Consultant
Robert Tomlin, DOD
Dr. Jerry Winkler, ASYSA Inc.

Standards and Environments Assessment Methodology Panel
Moderator: Richard Petersen, Capt. USN, Deputy Director
Information Processing Directorate, DISA Center for Standards

Panelists:

Dr Robert Balzer, Univ of So California, ISI;
Mr. Herm Fischer, Mark V, Chair, NAPI;
Mr. Herbert Kopplinger, Seimens Nixdorf, Munich, Germany;
Dr. Charles McKay, Univ of Houston-Clearlake;
Mr. Myer Morron, Independent Consultant, Chair, PIMB, Reading, England;
Mr. Robert Tomlin, DOD, ICASE Technical Director;
Dr Jerry Winkler, ASYSA, Inc., Chair, X3H4.

These are some of the questions to facilitate panel discussion and give rise to other questions and issues within the standards, environments, and processes domain:

1) What are the key issues to be addressed in selection of environments standards and related standards?

2) Is there an acceptable definition of a software environment?

3) On the premise that an environment assessment methodology can be defined, is a standards assessment methodology required? If yes, what are the key factors or components of such?

4) Should there be an environment assessment standard? If yes, do we define as a set of guidelines or a formal standard?

5) What are the key elements that a software environment must support? Are process standards one of those key elements? What is the relationship of environments, processes, and automation?

6) How do you assess the impact of new technologies on such things as environments? i.e. Domain specific architectures, re-use, concurrent engineering, middleware, business process re-engineering or enterprise modeling.

7) What are the forces in the commercial marketplace that will influence the composition of environments in the future?

HBCU Net Panel

Moderator: Shelton Lee, Jr., Major, USMC, DISA

Panelists: O. Diane Bowles, VITREC
Glen C. Moore, DISA
Ezekiel V. Salter, DISA
Tepper L. Gill, Ph.D., ComSERC
Al DeMagnus, Computer Management Services

Towards Development of an Integrated Telecommunications Network To Support Research and Community Based Learning for Historically Black Colleges and Universities and Minority Institutions

By Glen C. Moore: Director, Office of Small and Disadvantaged Business Utilization, Defense Information Systems Agency

Introduction

In 1991, the Defense Information Systems Agency (DISA), Office of Small and Disadvantaged Business Utilization (SADBU), embarked on a program to define the requirements and an architecture for a telecommunications network for Historically Black Colleges, Universities and Minority Institutions (HBCUs/MIs). Towards this end, DISA established a baseline and proposed a "Master Plan" whereby the nation's 100+ HBCUs/MIs would be able to participate in the mainstream of information technology. DISA's plan called for the establishment of a high speed voice, video, and data network to integrated various platforms and networks presently in operation at HBCUs/MIs. Dubbed HBCUnet, the architecture is taking a evolutionary approach to the development of an integrated network designed to bring minority institutions of higher education more fully into the mainstream of electronic educations and scientific exchange. Those institutions with federally funded R&D projects will particularly benefit from the improved connectivity and opportunities for information sharing which the evolving network will provide.

In accordance with Executive Order 123667 and Public Laws 99-661, 100-180, and 100-510, which espouse infrastructure assistance and increased participation of HBCUs/MIs in government sponsored programs, HBCUnet has as its objectives:

To increase communications between HBCUs/MIs and government agencies.

To provide information exchange between universities and government agencies for recruiting purposes.

To increase communication and cooperation among HBCUs/MIs by encouraging collaboration on federally sponsored R&D projects.

To provide a medium for the conduct of distance learning at the campus and community level.

To enhance the curriculum at HBCUs/MIs in the areas of engineering, computer science, and information technology.

Fundamental Requirements

The HBCUs/MIs have two fundamental telecommunications service requirements: Data Transmission Service (DTS) and Video Teleconferencing Service (VTS). These are the initial services which are envisioned for the evolving HBCUnet.

> The DTS links HBCUs and MIs, who are conducting research or other types of services for the DOD, with other institutions of higher learning, private industry, and other federally funded research centers and laboratories. The primary underlying network is DOD's Defense Data Network (DDN). Through the DDN the HBCUs/MIs will also have access to the Internet. Requirements for access to the DOD's Simulation Network (SIMNET) are currently under consideration. A concomitant requirement is for data transmission speeds on the order of 384 kbps and services including electronic mail, Bulletin Board, File Transfer, and Virtual Terminal.
>
> The VTS will provide interactive, point-to-point, and multipoint conferences in near full motion among HBCUs, MIs, and selected DOD agencies. Users will have rollabout video teleconferencing equipment or access to video rooms and will be able to hold conferences with sites on the DOD Defense Commercial Telecommunications Network (DCTN).

Existing Baseline

Three systems currently providing teleconferencing and video transmission capabilities for HBCUs/MIs are considered baseline systems upon which the HBCUnet will build:

> The CAmpus Relations Through Teleconferencing and Systems (CARTS) project is a network composed of seven (7) government and HBCUs/MIs sites providing services to 20 HBCUs using the SPRINT Meeting Channel (SMC). Each CARTS site has a dedicated T-1 connection to the SMC point of presence (pop), using Kentrox-T smart diagnostic T-1 communications service units (CSUs). The SMC provides connectivity to over 800 public and private video conferencing facilities worldwide. The SMC also provides all needed video conversions (CODEC and speed) and a Network Management Center for troubleshooting and problem resolution. In addition, the system includes an automated reservation center for conference and equipment setups. Each CARTS site is equipped with an Open Network Server (ONS) 400 high speed T-1 format processor collocated with a VideoTelecom Corporation System 300/350 CODEC. This configuration provides both video compression and a duplex audio system with adaptive echo cancellation on a PC platform. Additional conferencing equipment consists of the Videoconferencing System, Inc. Profile Unit and Master Conference unit providing color camera, monitoring, graphics and control capabilities. Augmenting the CARTS video conferencing feature is the CARTS Management Information System (MIS) which consists of data bases with information on student and faculty resumes and capabilities, employment

and student/faculty research opportunities. CARTS-MIS is hosted on an IBM 6250 RT and uses Informix SQL for database queries.

A T-1 gateway via the Virginia State University in Petersbury, VA provides connectivity from CARTS to the Black Colleges Satellite Network (BCSN). The BCSN primarily provides satellite downlink transmission of educational programming to over 100 sites, although a few sites have an interactive capability as well.

A second gateway is provided to the DISA managed DCTN which in turn provides connectivity to approximately 100 DOD videoteleconferencing sites.

Fundamental Considerations

In developing the future direction of HBCUnet some fundamental considerations arose, both technical and economic.

Motivation to Integrate Video and Data: Integration of services can be accomplished on a transmission and switching basis, or with any combination of these services at the user terminal facility. The current motivation for HBCUnet is, however, to integrate data and video transmission in order to reduce access line and intersite transmission costs. The problem generally experienced when attempting such consolidations is that the differing characteristics required for the DTS and VTS delay, and variable delay in particular, do not permit easy integration without sacrificing considerable flexibility and interoperability. Consequently, it appears that the integration issue may be postponed until integrated data/video terminals are realized and the VTS and DTS are viewed as integrated rather than separate services.

Preserving Current Terminal Equipment Investments: Large investments have been made by the HBCUs/MIs in video teleconferencing equipment and it makes little economic sense to force replacement/upgrading to meet a new transmission architecture unless a quantum change in service characteristics and levels can be assured. Thus, in building HBCUnet in its initial phases, maintaining compatibility with existing terminal equipment will be a major factor. Use of the existing BCSN is also being seriously considered, however, because most nodes on this network have receive only capability, conversions of these systems to allow interactive videoteleconferencing need to be explored.

Migration to National/International Standards: As the emerging CCITT video and video compression standards are realized in hardware/software, there will be pressures to convert existing non-standard equipment. For example, the DCTN is migrating from its current Compression Labs Incorporated (CLI) Rembrandt CODECs to ones that are CCITT H.320 compliant, which will require the CARTS to do likewise to maintain interoperability. Replacement of existing CODECs used in CARTS/BCSN to software programmed units which would interoperate with the DCTN CCITT compliant CODECs would have the added advantaged that future upgrades could be accommodated in software.

Formation of HBCUnet

HBCUnet is being designed to fulfill the requirements outlined above to some 20 initial nodes. Equipment to establish the nodes will be obtained, in part, through DOD's Automation Equipment Program sponsored out of the Defense Automation Resource Information Center (DARIC) a division of DISA's Center for Information Management (CIM). Major information technology (IT) providers to the DOD, to include telecommunications carriers and systems integrators are in the process of forming partnerships with the HBCUs and MIs involved in the project.

Minority owned businesses have played a key role in the HBCUnet project to date, donating equipment, personnel, financial resources, and services to HBCUnet participants.

Two of the near term requirements of HBCUnet is to provide access to the DDN/Internet and interconnectivity to the DCTN. Currently only a few HBCUs/MIs have DDN connectivity. As a near term action, participation of these institutions in the Internet community via the DDN will be significantly increased. Consideration is being given to allowing some HBCUs/MIs which are in close proximity to existing DCTN sites to share video teleconferencing facilities with those DOD organizations.

Proposed HBCUnet Architectures

The network management site for voice and data for the network, as well as continued research and development, will be Howard University's Computational Sciences Research and Engineering Center (ComSERC). Clark Atlanta University will act as the HUB site for videoteleconferencing. The following architectural approaches to HBCUnet have been proposed by Howard and Clark Atlanta Universities:

IDNX NETWORK

Howard University has proposed an Integrated Digital Network Exchange (IDNX) architecture in a star configuration which consists of a hub at Howard comprised of a Synoptics concentrator, data transfer equipment, a Network Equipment Technologies (N.E.T.) IDNX-70 transmission resource manger, and DSU/CSU. Outlying HBCUs/MIs nodes would have Synoptics concentrators, data transfer equipment, and IDNX-20 transmission resource manager, DSU/CSU and be linked to the hub at Howard via T-1 circuits. A dedicated T-1 would also link the hub to the DOD DCTN. The hub computer would be an Alliant FX/2800 Super Computer with HIPPI interface and 50 GB file server. Howard is currently expanding this proposal to include a 56 kbps X.25 packet switching capability.

DUAL 56 NETWORK

Clark Atlanta University, the manager of the CARTS project, has proposed a teleconferencing network using dual 56 kbps circuits with switching provided by SPRINT's dual 56 Video Teleconferencing Network (VTC). With the hub located at Clark Atlanta, management of VTC will be centralized. The Clark Atlanta HUB would maintain T-1 connectivity to the SMC and the DCTN. The video equipment at the hub and the outlying nodes would consist of video teleconferencing equipment, CCITT H.261 compliant CODECs, and various controllers. A CCITT H.221 compliant digital video branch exchange would provide for switching at the hub an interface with like compliant multiplexer/modems at the sites. DDN Internet access would be provided by separate dial-up lines.

Refinement studies are underway to examine other approaches which might best exploit or combine the features of these proposals. The first is a combination of the two with both IDNX and packet switching capabilities hubbed at Howard. The video teleconferencing portion would be similar to that proposed by Howard. In addition, Howard would also serve as one of three hubs for the DTS provided on a 56 kbps packet switched network. The other hubs would be located at Fort Valley State College and St. Augustine's College.

The second approach is a multi-hub network with IDNX hubs at Howard, Southern University and Atlanta University Center serving both VTS and DTS requirements. Dual T-1 Connectivity to SMC would be provided form Howard and Atlanta.

Studies are also under way to examine alternative transmission services for the HBCUnet. These include Integrated Services Digital Network (ISDN) offerings which will be available at many HBCUs/MIs sites next year, bandwidth-on-demand, and LAN interconnect services such as frame relay and switched multi-megabit data service (SMDS).

Conclusion

As our nation moves towards a comprehensive information technology infrastructure, DISA is proud to be the principal partner in what could very well be considered proof of concept of a "National Information Infrastructure". HBCUnet holds significant promise as the principal vehicle in making the nation's HBCUs/MIs full fledged partners in the cooperative government/educational R&D community. Plans are underway to designate HBCUnet as one of several national testbeds of the National High Performance Computing Program. Industry will gain from alliances with HBCUnet participants and HBCUs/MIs in general, as these institutions are to only fertile grounds for recruitment of minority engineering talent, but also offer unique procurement opportunities in areas set aside for their participation.

Expanding cooperative research and development partnerships between government and educational communities has proven to be beneficial for the country as a whole, while ensuring our continued competitiveness in the global marketplace.

DOD Open Systems Environment Guidance Panel

Moderator:	Huet Landry, DISA Center for Standards
Panelists:	John Stanton, DISA Center for Standards Brian Purdy, DISA/JIEO/TFCC Gary E. Fisher, NIST

DOD Information Technology Standards Guidance

Mr. John Stanton
DISA Center for Standards
DISA/TBEB
11440 Isaac Newton Sq N. Ste 210
Reston, VA 22090-5006
voice: 703-487-8407
e-mail: stantonJ@cc.ims.disa.mil

Gary E. Fisher
NIST, Technology Bldg. Room B266
Gaithersburg, MD 20899
voice: 301-975-3275
e-mail: gfisher@nist.gov

Mr. Brian Purdy
DISA/JIEO/TFCC
701 S. Courthouse Rd.
Arlington, VA. 22204-2199
voice: 703-696-8856
email: purdyb@cc.ims.disa.mil

Abstract

The Defense Information Systems Agency (DISA) Center for Standards has embarked on an ambitious project to define the DOD Open System Environment (OSE) and, over time, to populate that environment definition with standards to satisfy OSE requirements. This paper describes the Information Technology Standards Guidance (ITSG) and how it can be used by architects and program managers to develop a profile of standards to meet their requirements. The ITSG identifies formal and emerging standards, public domain specifications, and the interrelationships among the recommended standards. It provides useful information on the recommended standards for each service area such as portability guidance and the recommended standard usage. It includes recommendations that can be used in acqusitions today, as well as guidance on future standards. The ITSG guides DOD Program Managers (PMs), system engineers, and architects in acquiring and using of standards-based products.

1.0 Background

One of the lessons of Desert Shield/Desert Storm was the need for greater interoperability through standards. At the same time, decreasing budgets have brought interest in the use of "dual-use" commercial standards to reduce cost and improve competition. The Defense Information Systems Agency (DISA) Center for Standards (CFS) was created to coordinate and manage the development of standards across the DOD and to produce DOD-wide standards guidance.

DOD Information Technology standards guidance was originally within a chapter of the DOD Technical Reference Model (TRM). The TRM was based on work done at the National Institute of Standards and Technology (NIST). The Center for Standards, with the participation of Defense Services and Agencies, developed the Information Technology Standards Guidance (ITSG) based on the service model in the TRM.

The ITSG expands each service area in the TRM to a finer granularity. It then provides guidance on selecting information technology standards to fulfill OSE base service area requirements. The ITSG is a combination of the TRM standards guidance, the DOD OSE definition and the standards guidance previously found in the Open Systems Environment Profile for an Imminent Acquisition (OSE/IA). The ITSG specifies the DOD base standards that are the basis for standardizing profiles across the entire department.The ITSG addresses the needs for near-term acquisition guidance and provides significant additional information on imminent acquisitions. The information within the ITSG is the first major step for satisfying the goal for a DOD consensus, open systems, target environment definition and the environment's information technology standards.

2.0 Technical Architecture Framework for Information Management

The TAFIM is an enterprise-level[1] guide for developing technical architectures to satisfy specific functional requirements. It is assumed that all information systems must interoperate at some time. Therefore, their architects and designers should use the TAFIM as the basis for developing a common target architecture to which systems can migrate, evolve, and interoperate. Over time, interoperability between and among the number of systems will increase, providing users with improved services needed to achieve common functional objectives. To achieve portability, standard interfaces will be developed and implemented. Scalability will be developed in mission applications to accommodate flexibility in the functionality. Proper application of the TAFIM guidance can: promote integration, interoperability, modularity, and flexibility; guide acquisition and reuse; and speed delivery of information technology and lower its costs. The TAFIM consists of eight volumes:

Volume 1: Introduction
Volume 2: Technical Reference Model
Volume 3: Architectural Concepts and Design Guidance
Volume 4: Implementation Guide (Standards-Based Architecture Planning Guide)
Volume 5: TBD
Volume 6: DOD Goal Security Architecture
Volume 7: Information Technology Standards Guidance
Volume 8: DOD Human-Computer Interface Style Guide

[1] This should be read as Departmental- or DOD-Level, which are synonymous with enterprise-level.

The DISA Center for Architectures is responsible for the production of the TAFIM. The current Version 1.2 of the TAFIM was signed out by ASD(C3I) in January 1993. Version 2.0 has been distributed for coordination and comment, and is expected to be published in Spring 1994. For more information, contact Mr. Brian Purdy at (703) 696-8856, or send e-mail to TAFIM@mitre.org.

3.0 Open Systems Environment

3.1 OSE Principles

The fundamental principle on which the ITSG is based is that an Open System Environment is defined by the set of basic technical services required to support the functional requirements of the environment. Some definitions of an OSE are nothing more than a list of base standards. This is insufficient for several reasons. First, there are many base services for which standards have not been developed. There are also many required functions that are not defined by the base standards, as well as options that are not specified within the base standard. Industry groups are working outside the formal standardization committees to define specifications to fill some of these gaps. These specifications will be missing from a list of base standards. Finally, there is no way for the program office or systems integration personnel to verify the completeness of a standards profile without an appreciation of the services that are supplied by the standards.

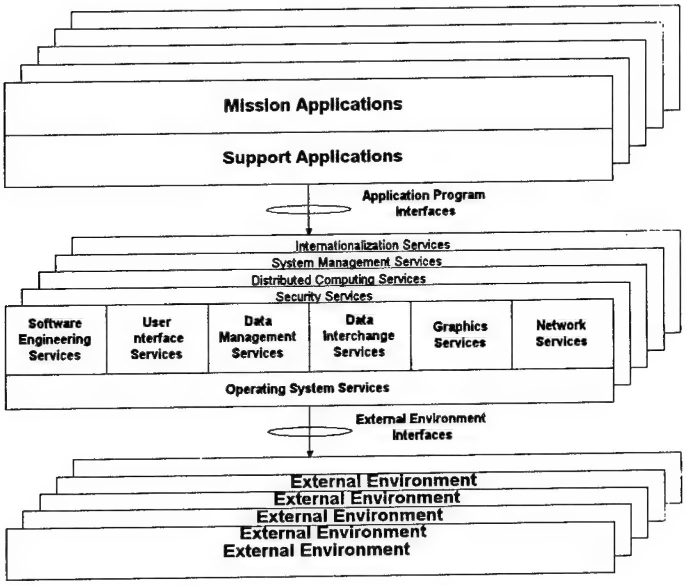
Information technology standards are layered between the system architecture and the platform or application environment. A well conceived architectural framework will list its functional requirements. The ITSG specifies standards intended to meet these architectural functionality requirements.

3.2 Base Service Areas

The ITSG is organized according to the DOD Technical Reference Model (TRM), pictured in Figure 1. This model is based on POSIX, and identifies three types of entities from a computer perspective: applications, platforms and platform services, and the external environment. Platform services are divided into eleven service areas, as shown. The ITSG expands on these service areas.

Each major service area is decomposed to the base service area level, containing base standards. The decomposition of a major service area results in dozens of smaller service areas called OSE base service areas. There are additional services below the base services, but the ITSG does not attempt to represent them. The ITSG includes all base service areas currently envisioned as part of the OSE. However, the set of base service areas will expand over time as additional functionality is required. The ITSG has over 300 OSE base service areas. For ease of reference, the base service areas are grouped into mid-level service areas.

Figure 1: DOD Technical Reference Model



For example, the Data Management Services major service area breaks down into mid-level service areas as follows:

a. Basic database services.
b. Information data dictionary/directory services.
c. Distributed data.
d. Object database.
e. Transaction processing.

Within the mid-level service area called "basic database services," the ITSG contains the following OSE base service areas:

a. Data definition, manipulation, and query.
b. Data integrity.
c. Embedded Structured Query Language (SQL).
d. Dynamic facilities.
e. Indexed sequential access.
f. Forms query and management.
g. Report writer.
h. Database administration.
i. Menu-driven database access.
j. Data storage and archiving.
k. Database security.

An example of one of these OSE base service areas follows in section Attachment 1.

The TRM and ITSG together comprise a definition of the service areas supported within the Information Technology domain detailed down to the base service areas. These base service areas define fundamental functions. Through the process of standardization upon a consistent and stabile framework of base service areas it becomes possible to compare and contrast competing standards. The aggregate definition of functionalities described by the base service areas becomes the DOD definition of an Open System Environment. This Open Systems Environment provides the consistent framework for describing functional requirements, assessing standardization needs, and supporting development of profiles.

The goal of the ITSG, which will never be fully realized, is to define standards for all of the base service areas. The ITSG provides information and guidance for all areas of the OSE as currently known.

4.0 Base Service Area Descriptions and Standards Guidance

4.1 Overview

The ITSG is published in two parts. The first part, which is TAFIM Volume 7, contains an overview, the DOD Enterprise-level Profile (discussed in Section 5 of this paper), and guidance on how to apply the ITSG as a whole. The second part of the ITSG is published as a separate document. It describes the functional characteristics of the DOD OSE at a high level, then presents guidance on profiles. The majority of its content are detailed service descriptions and recommendations, which were originally published as the OSE/IA. The remainder of paper is focused on this latter material.

4.2 Base Service Descriptions

The OSE base service area descriptions are the focus of the use of the ITSG. An OSE base service area is a logical entity within the OSE that requires some form of standards solution. Some standards logically fall within a specific OSE base service area. In other cases, different OSE base service areas sometimes list the same standards because a particular standard satisfies more than one OSE base service area requirement. For each OSE base service area there is a brief definition of the OSE base service area, and the following topics are addressed:

3.X.Y.Z.1 Standards.
3.X.Y.Z.2 Standards deficiencies.
3.X.Y.Z.3 Portability caveats.
3.X.Y.Z.4 Tailoring guidance.
3.X.Y.Z.5 Related standards.
3.X.Y.Z.6 Recommendations.

Standards. The first sub-part of the OSE base service area description is the standards table. The standards table is the key to using the ITSG. These tables include all applicable standards and specifications satisfying the base service area and organizing them in the prescribed DOD standards hierarchy. The intent is to capture all the standards conceivably useable within DOD because of the non-open legacy systems that will be migrating toward open systems may implement standards other than the recommended one. The standards table in each base service area has several features requiring further explanation.

The top row of the standards table contains the current DOD OSE target standard satisfying that particular OSE base service area. The remaining parts of the base service area description provide additional necessary information about the target standard and the other standards listed in the table. Many blank top rows are found in standards tables throughout the ITSG. A blank top row indicates no formal standards are available to meet the functionality requirements. If a top row is blank, choose another suitable standard specified in the lower rows, in the order of appearance in the table. Also, the text accompanying the base service area contains further needed guidance for a solution.

The criteria for selecting the preferred standard place greatest emphasis on the openness of the specification, which includes its public availability and its control through a public, consensus process. They also include legal and functionality considerations. Industry support and commercial availability are also strong considerations. Preference is given to international and national voluntary standards before government and DOD standards when military requirements are met.

The extensive information about the standards population within a specific base service area provides a full perspective of the activity in that area. The standards in the lower rows are provided because for many reasons, some acquisitions cannot use the standard specified in the top row. For example, they may be transitional systems in which portion of a legacy system will remain temporarily unchanged. If a standard listed in a lower row is specified in an acquisition, then the acquired system will diverge from the target DOD OSE. Standards listed in the lower rows of the tables diminish the probability of achieving portability and interoperability and/or increase the ultimate life-cycle cost required to achieve an Open System state.

The standards tables of some base service areas show what may be considered a paradoxical situation. There may be more than one "top row" standard. This occurs, for example, in the geodesy area where different data sets are appropriate to different map scales. This situation shows that more than one standard may be preferred depending on specific architectural requirements.

All the **Standards** are combined under 3.X.Y.Z.1 into a single table. The first column of the table lists the type of standard or specification. Program managers can select standards and specifications based on the preference hierarchy. The second column lists the sponsoring organization of the standard (e.g., NIST is the sponsor for a government standard for the Computer Graphics Metafile (FIPS-128)). The third column lists the name of the standard. It also mentions any matching standards (see section 2.3.1.2, Matching standards, above). The fourth column lists the numerical designation of the standard (e.g., the NIST FIPS for Structured Query Language is 127-1). The last column lists the status of the standard (i.e., the state of completion), where "IS" means International Standard.

Following each standards table are six additional categories giving additional explanation of the standards listed in the table. The standard listed in the top row (the recommended standard) is given primary emphasis. The associated text is intended to support primarily the recommended standard. Information about the remaining standards provides assistance for the DOD legacy systems that may continue to use other standards during their transition to the DOD OSE target environment. An example of associated text that corresponds to the example standards table follows in Attachment 1.

Standards deficiencies identifies deficiencies in the standards and recommends how to apply the standard to reduce their impact. It addresses known problems within the standards such as missing features. In the standards deficiencies example in Attachment 1, several problems with the existing standards have been identified. Because the standard is new, no deficiencies in FIPS PUB 127-2 have been identified.

Portability caveats addresses the features of the standard hindering portability. The portability caveats example points out particular problems of the SQL standards: implementation-defined exception code values and the character data type. Additional portability problems arise between the NIST FIPS for SQL and the other versions of the standard as shown in the text.

Tailoring guidance gives additional guidance about selection of options and features of a standard. Tailoring guidance also includes solutions to the portability problems identified above to enhance portability and overcome problems.

Related standards addresses the standards required as a foundation for a particular standard, or other standards relating to the functionality under discussion, or other interfacing standards. A prime example of this would be IEEE P1003.1 (POSIX.1) as a related standard for using the coming IEEE P1003.4, which is the real time extension to the POSIX.1 standard. In the related standards example in Attachment 1, standards usable to extend SQL functionality have been identified. These standards include RDA, and all may be found in the standard tables of other base service areas.

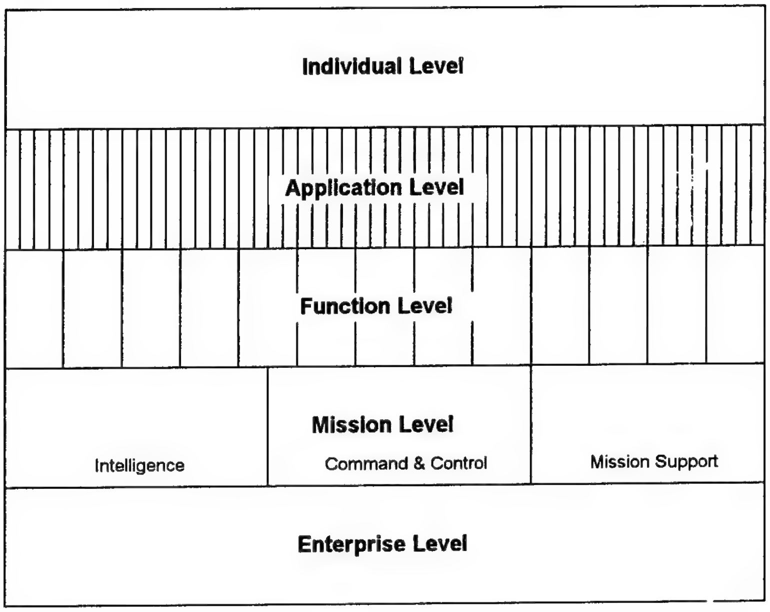
Recommendations advises which standard is preferred for specification in the procurement for the particular area of functionality and standards. The recommendation will provide suggested wording to use in the procurement when possible. In the example recommendations, the most current SQL and supporting standards are recommended along with details about optional conformance levels and testing.

5.0 Application of the ITSG

5.1 DOD Standards Profile

The TAFIM includes a discussion of the DOD Integration Model, illustrated in Figure 2. This model represents the fact that the DOD Enterprise is divided into a few Mission areas, each supporting multiple Functions. Applications are developed to support one or more functional areas. Each level is built on the guidance and standards specified at the supporting levels. Standards guidance for the DOD and each Mission, Function, and Application system are contained in profiles. A profile identifies the relevant standards, along with any required subsets, options, or parameters, required to support the specific level.

Figure 2: DOD Integration Model



The ITSG contains the DOD Enterprise Level profile of standards, also referred to as the DOD Profile. It supports the OSE base services that are presumed to apply across the preponderance of systems within the DOD. The profile identifies the DOD target standards for these service areas.

5.2 Mission, Function, and System Profiles

The DOD Profile is the basis for developing profiles for Mission/Functional areas or systems. The architect or program manger will identify the OSE functions and services required to support the area or system. The ITSG can be used to identify standards and specifications to satisfy those services that are not covered by the DOD profile. The architect or program manager should also identify the specific subsets, options, and features required, and document them in the profile. The ITSG contains additional guidance on the development and documentation of profiles.

6.0 Status of the ITSG

6.1 Current Status

The ITSG was distributed for Service and Agency comment in December 1993, using the distribution list for standards in the DCPS, IPSC, and INST standardization areas. It was also distributed in January 1994 as part of the TAFIM review. Part one of the ITSG will be published along with the TAFIM Version 2.0 in Spring 1994. It should be available by the time this paper is published. Part two of the ITSG will be published by the Center for Standards at about the same time.

6.2 Future plans

The technology within the focus of the ITSG is growing and changing dynamically. Additionally, the standards organizations are actively adding to the body of consensus based standardization information. The emerging internationalization of information technology requirements is stimulating both harmonization and acceleration of standardization activity to accommodate compatibility and competitiveness in the world arena. To meet the challenges of the fast-paced information technology domain and the decentralized decision-making essential to the execution of DOD programs, the ITSG must evolve at a pace consistent with both requirements. It will be published on an regular cycle.

The next major version of the ITSG is scheduled for publication in Fall 1994. The document will undergo a significant edit to reduce duplication. It will be expanded to incorporate additional services and functions, and possibly additional major service areas. The recommendations will be enhanced to provide additional suggested RFP language. The CFS is also developing a database of ITSG information to make the material more readily available to architects and program office users. The goal is to supply the ITSG information on a CD-ROM with flexible retrieval software.

Working groups of subject matter experts will be used to develop the recommendations within each major service area of the ITSG and to evolve the subject area coverage. DOD personnel are encouraged to support these working groups. If interested in participating, please contact the appropriate CFS subject area point of contact identified in Table 1 below.

7.0 Conclusion

The Information Technology Standards Guidance (ITSG) is a tool. Its purpose is to define the Department of Defense Open Systems Environment and provide guidance to meet the requirement for consistent standards selection for profiles for DOD IT acquisitions. It does this by defining the DOD OSE and the target group of IT standards that DOD systems are to use in implementations. The specification of the DOD Open System Environment and the IT standards contained within that environment using the ITSG will guide system convergence toward the DOD consensus target environment.

The consumer of ITSG information is encouraged to contact CFS for assistance or to identify functional requirements and/or standards not yet incorporated into the document. The CFS will appreciate additional inputs on the use of standards, deficiencies and future needs. For additional information on the ITSG, or to obtain a copy., contact the Directorate for DOD Standards Assistance (DISA/TBxx), Center for Standards (CFS), Joint Interoperability and Engineering Organization (JIEO), Defense Information Systems Agency (DISA), 11440 Isaac Newton Square North, Suite 210, Reston, VA 22090-5006, or call (703) 487-8296.

TABLE 1. Service Area Points of Contact

Section	Service Area	CFS POC (Name, Office Code)	Phone (Area 703)
	Introduction/Guide	Mr. John Stanton, TBEA	
3.2	Programming/ Software Engineering	Mr. Paul Bacarro, TBEB	487-8422
3.3	User Interface	Mr. John Sarkesain, TBEA	487-8407
3.4	Data Management	Dr. Dan Wu, TBEC	487-8409
3.5	Data Interchange	Mr. Alan Peltzman, TBEC	487-8409
3.6	Graphics	Mr. Alan Peltzman, TBEC	487-8409
3.7	Networking	Mr. Walt Luchessi, TBBD	(908) 532-7726
3.8	Operating Systems	Mr. Steve Law, TBEA	487-8407
3.9	System Management	Mr. Larry Spieler, TBEA	487-8407
3.10	Security	Mr. Bill Smith, TBED	487-8252

Attachment 1: Sample ITSG Base Service Area Entry

3.4 Data management services.

3.4.1 Basic database services. These standards provide the basic database services needed by an application using a database. A database management system (DBMS) is an application used to create, store, retrieve, change, manipulate, sort, format, and print the information in a database.

3.4.1.1 Data definition, manipulation, and query. Data definition includes create, alter, and delete tables, views, records, fields, classes, objects, instances, attributes, and data. Data manipulation includes insert, select, update, and delete tables, views, records, fields, classes, objects, instances, attributes, and data. Data query includes the ability to specify search conditions consisting of a combination of select lists, predicates, and comparison operators.

3.4.1.1.1 Standards. Table 3.4-1 presents standards for data definition. manipulation and query services.

TABLE 3.4-1. Data definition, manipulation, and query standards

Standard Type	Sponsor	Standard	Standard Reference	Lifecycle Status
Government Public Consensus	NIST	FIPS Database Language SQL (adopts ANSI X3.135-1992) (same as ISO/IEC 9075:1992)	FIPS 127-2:1993	Approved (Mandatory)
International Public Consensus	ISO/IEC	Database Language SQL (same as ANSI X3.135-1992)	9075:1992	Approved
National Public Consensus	ANSI	Database Language SQL (same as ISO/IEC 9075:1992)	X3.135-1992	Approved
Government Public Consensus	NIST	Guidelines for Functional Specifications for Database Management Systems	FIPS 124:1986	Approved
International Public Consensus	ISO	Database Language - Network (NDL)	8907:1987	Approved
National Public Consensus	ANSI	Database Language - NDL	X3.133-1986	Approved
Government Public Consensus	NIST	Database Language - NDL (adopts ANSI X3.133-1986)	FIPS 126:1987	Approved
Government Public Consensus	NIST	FIPS Database Language SQL (adopts ANSI X3.135-1989 and X3.168-1989) (same as ISO 9075:1989)	FIPS 127-1	Superceded
International Public Consensus	ISO/IEC, ANSI	Database Language SQL3 (will replace SQL2)	9075 (future), X3H2 Project 0525-R	Emerging (IS: 1995)
Government Public Consensus	NIST	FIPS (Based on ISO SQL3)	FIPS 127-3 (future)	Future (Goal 1995)

3.4.1.1.2 Standards deficiencies. The following deficiencies in the standards have been identified:

a. International Organization for Standardization (ISO)/American National Standards Institute (ANSI) SQL and National Institute for Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 127-1 are designed for stand-alone, single environment databases. Interactive SQL is specified by FIPS 127-1, but not by ISO/ANSI SQL1.
b. There is no standardized way to specify logical database access control, which is important to database security.
c. Hashing methods to access data are not standardized or in progress.
d. SQL1 is inadequate and has failed to be transportable or standardized to be very useful. The upcoming SQL-3 provides an opportunity for Department of Defense (DOD) requirements to be inserted.

3.4.1.1.3 Portability caveats. SQL 2's segmentation into multiple levels increases the likelihood of incompatibility between different vendors' SQLs because different vendors will implement entry level SQL 2 and then choose options from other levels.

The ISO, ANSI, and FIPS versions of SQL specify state exception code values (called SQLCODE parameters) such as 0 for successful execution, 100 for nonexistent data, and implementation defined code values for particular exception conditions. Different products that conform with SQL have different SQLCODE values for exception conditions. The set of SQL character values for the character data type and collating sequence of characters is defined by the implementor, and therefore, nonstandard in products. FIPS 127-1 specifies a number of sizing constraints for database constructs. These are implementation defined in ISO and ANSI SQL, and consequently, differ in conforming SQL products. Also, ISO/ANSI Level 1 SQL is minimal, not testable, and disallowed by FIPS 127-1 which requires the higher levels of compliance to the ISO/ANSI standard.

3.4.1.1.4 Tailoring guidance. Specify SQL2 (and later SQL3) as soon as possible because SQL2/3 contains greater standardized functionality than SQL1. This will reduce the use of non-standard extensions. SQL2 also standardizes more than 60 SQLCODE exception code values.

All products that conform with SQL or comply with FIPS 127-1 contain nonstandard, incompatible options. To ensure portability when using SQL1, specify and use the FIPS 127-1 optional flagger to flag the nonstandard options so that developers can write conforming code.

Carefully specify and check all sizing constraints for a procurement to meet functionality requirements and avoid portability problems.

Avoid the Network Data Language (NDL), if possible, because it is little used and will not be upgraded.

Specify the ISO Remote Data Access (RDA) standard, and also the X/Open or SQL Access Group's RDA and Call Level Interface (CLI) specifications, in conjunction with SQL/SQL2, in order to obtain remote data access capabilities in a distributed environment.

3.4.1.1.5 Related standards. The following standards are related to data definition, manipulation, and query:

a. ISO draft international standard (DIS) 9579: Remote Database Access (Generic RDA) (supports remote database access in client-server environments) (IS expected in fiscal year 1993 (FY93))
b. ISO DIS 9579-2 (SQL Specialization)
c. SQL Access Group's SQL Access Formats and Protocols (FAP) (1991)
d. SQL Access Group's Call Level Interface (CLI)
e. X/Open Remote Data Access (RDA) Preliminary Sepcification (Identical to the SQL Access Group's RDA Specification
f. X/Open's Call Level Interface (CLI) Snapshop Specification (Identical to the SQL Access Group's CLI Specification)

3.4.1.1.6 Recommendations. Consult the wording suggested in the October 1991 GSA publication for proposed language for requiring that a database conform to SQL, and consult FIPS 127-1 for guidance on how to structure a Request for Proposal (RFP). The FIPS "flagger" (to flag nonconforming extensions) is optional and must be specified explicitly.

If interactive SQL is required, a procurement must indicate explicitly whether or not "direct invocation of SQL statements" is required and, if required, which SQL statements are to be directly invocable. If not specified, the default is "CREATE TABLE," "CREATE VIEW," "GRANT privilege," "SELECT" with "ORDER BY" option, "INSERT," "UPDATE:searched," "DELETE:searched," "COMMIT WORK," and "ROLLBACK WORK."

Explicitly specify sizing constraints for database constructs. The FIPS 127-1 sizing specifications are reasonable to expect vendors to deliver, but are fairly minimal. Since database construct sizing specifications depend on the procurement, a procurement can override them.

Require the use of NIST conformance tests and/or services to validate conformance to the SQL-based FIPS for required and optional FIPS 127-1 features. Testing applies only to a specific platform, so call for conformance tests for each platform bid. Use the quarterly list of processors validated against FIPS 127-1 and 127-2 by NIST to help evaluate bids.

Specify the NIST's Transition Level SQL 2 and the SQL Access Group's (SQL) CLI and RDA interfaces and protocols for the following reasons. Most DBMS vendors have no intention of conforming to the FUll Level SQL 2:1992 because SQL 2 Full Level is very large and complex. As a result, the time it will take to add the necessary features will likely exceed the time before the SQL 3 standard is completed. To ensure portability, as well as functionality, it is recommended that users include the following two specifications in their procurements:

a. NIST's Transition Level SQL 2 (specified in FIPS 127-2), which is a hybrid of Entry Level and higher levels of SQL 2:1992.
b. SQL Access Group's and X/Open's (SAG) Call Level Interface (CLI) and Remote Data Access (RDA) standards. The SAG specifications are not segmented like SQL '92 and

offer a nice balance between the Full Level SQL '92 feature set and what users need now. The SAG specifications include connection management acapabilities (which are part of the SQL '93 Full Level), schema manipulation and the CHARACTER VARYING data type (both of which are part of SQL '93 Intermediate Level), and features not included in any level of SQL '92 conformance, including the CREATE INDEX and DROP INDEX statements. SAG's specifications are published jointly with X/Open as X/Open specifications.

AdaSAGE Panel

Moderator: Joan McGarity, Naval Computer and Telecommunications Command

Panelists: Howard D. Stewart, Idaho National Engineering Laboratory
Kenneth D. Russell, Idaho National Engineering Laboratory
Paul H. Whittington, Idaho National Engineering Laboratory

AdaSAGE™ Notes

Howard D. Stewart
Kenneth D. Russell
Paul H. Whittington
Idaho National Engineering Laboratory
Special Applications Unit
P.O. Box 1625, Idaho Falls ID 83415-1609

January 19, 1994

Abstract

AdaSAGE is an application development tool for Ada programmers implemented as a set of Ada packages and a set of executable programs used as support utilities during application development and operation. Many questions have been asked regarding the characteristics of AdaSAGE (i.e., Is AdaSAGE relational?). Herein several such questions are posed and answered.

What is the purpose of AdaSAGE?

AdaSAGE was designed to provide tools and an environment for Ada programmers to develop major non-proprietary systems completely in Ada that are as good as or better than systems developed using alternate methods.

Who uses AdaSAGE?

The SAGE system began at the Idaho National Engineering Laboratory (INEL) in 1982. In 1987, it was made available in the Ada programming language for use on U.S. Marine Corps projects. Since then, it has been used by all of the major services of the DOD (Department of Defense), the DOE (Department of Energy) and private industry. It is available through a variety of distribution channels to both government, academia and private industry.

Where can one get AdaSAGE?

At the present time, AdaSAGE may be acquired from one of the following sources:

AdaSAGE Users' Group
Idaho National Engineering Laboratory
(208)526-0656

RAPID Center
Sgt. Stephen St. Espirit
(703)285-9007

Office Of Scientific Technology Information
P.O. Box 62
Oak Ridge, TN 37831
(615) 576-1166

Is AdaSAGE written in Ada?

AdaSAGE reusable Ada packages are 100% Ada code. In the MS-DOS environment, approximately 5% of AdaSAGE is written in assembly language to provide optimal interface to low level hardware that is not readily available to MS-DOS.

How extensive is AdaSAGE?

During AdaSAGE development, extensive effort is expended to create application development tools and associated system development methodologies that maximizes the effectiveness and productivity of Ada system developers.

The AdaSAGE system is comprised of over 300,000 lines of code representing over 250 person-years of effort with an average cyclomatic complexity of 4, as measured by a standard code metric analysis tool.

Does AdaSAGE support the Open System Environment?

The AdaSAGE development team is committed to supporting the Open System Environment standard. At the current time AdaSAGE provides support in the following ways:

Ada	AdaSage is written in Ada.
SQL	Interactive and imbedded SQL is provided.
GKS	Standard GKS is provided.
POSIX	AdaSAGE is available on POSIX systems.
GOSIP	AdaSAGE supports multi-user development on GOSIP networks.

On what platforms is AdaSAGE available?

Versions of AdaSAGE tools and reusable packages are available within a variety of environments. AdaSAGE is now available or will be available soon on the following hardware platforms:

Intel 8088
Intel 8086
Intel 80286 Real and Protected Mode
Intel 80386 Real and Protected Mode
Intel 80486 Real and Protected Mode
AT&T 3B2
Sun SPARC
RS/6000

AdaSAGE runs under the following operating systems:

MS-DOS
UNIX
AIX

The AdaSAGE application development system is available for the following Ada compilers:

Alsys
Meridian
Verdix

Can AdaSAGE be used to develop large systems?

Because AdaSAGE is written in Ada, it supports the development of very large systems. In practice, it has been used for the development of small to very large systems, supporting both large program and capacity requirements. Some of the capacity constraints are shown in the following table.

Category	Constraint
Relations (flat files) per data dictionary	1,000
Attributes per relation	500
Indexed attributes per relation	500
Forms per data dictionary	6,550
Record (tuple) size (bytes)	32,000
Field (attribute) size (bytes)	
Fixed length	32,000
Variable length	2,147,483,000
Significant digits per number	60
Records (tuples) per relation	2,147,483,000/Record Size
Name length (characters)	312

Are training and support available?

Beginning, advanced, and graphics training courses are available at the INEL or at a requested location if a sufficient number of students are available. The AdaSAGE Users' Group supports registered users with a technical support line, new release upgrades, a bulletin board, a newsletter, and other training and assistance. For more information, contact the AdaSAGE Users' Group at (208)526-0656.

Is AdaSAGE maintained?

AdaSAGE is supported by the INEL (a DOE laboratory) with funds supplied by the Ada Joint Program Office (AJPO) and other DOD agencies. A long term agreement is in place between the DOE and DOD for continual support in the future. Work is currently underway for conversion to other computer platforms, open system architecture adherence, and other areas.

Does AdaSAGE have a Data Dictionary?

A data dictionary contains the data structure (schema) and often other ancillary information about the data and its structure. AdaSAGE supports an independent data dictionary and a facility for defining, modifying, and reporting on this information. This capability allows many different applications to access the same data dictionary and provides physical and logical data independence as specified in rules 8 and 9 of Codd's Rules.

What security support is included in AdaSAGE?

AdaSAGE provides many levels of security. Even though data integrity checks are a form of security and provided by AdaSAGE, they will not be discussed here. Access security is provided by password protection at the database, relation (flat file), and the individual attribute (column) level. These are defined within the data dictionary and enforced throughout the system for both programmers and users. Additional security is often required within applications. Many such applications have been provided using AdaSAGE with additional protection through user tables that contain user identification information, statistical information, and network and system information concerning user logon identifiers and privileges.

Does AdaSAGE support the development of relational databases, and if so does it support Codd's rules?

One of the many features of AdaSAGE is its support for relational database development. E.F. Codd, a mathematician, first defined the relational rules that were presented in a paper published in ACM Communications, June 1970. Later, in 1985, he summarized a foundation principle and twelve rules that should be adhered to by a relational system. He stated that a database system fulfilling at least half of these rules and having the capabilities eventually to fulfill all of them can be considered relational. The following table indicates the adherence to these rules by AdaSAGE, ANSI SQL (the source for FIPS 127 and FIPS 127-1), and IBM's DB2, which is considered by Mr. C. Date of the Relational Institute to be the leading SQL representative database.

Codd's Rule	Description	AdaSAGE	ANSI SQL	IBM DB2
0	Foundation Principle	P	P	P
1	Information Rule	Y	Y	Y
2	Guaranteed Access Rule	Y	N	P
3	Missing Information Rule	P	N	P
4	System Catalog Rule	P	N	Y
5	Comprehensive Language Rule	P	P	P
6	View Updatability Rule	N	N	P
7	Set Level Updates Rule	P	P	Y
8	Physical Data Independence Rule	Y	Y	Y
9	Logical Data Independence Rule	Y	P	P
10	Integrity Independence Rule	P	P	P
11	Distribution Independence Rule	P	?	IS
12	Nonsubversion Rule	Y	?	Y

Y = Yes N = No P = Partial IS = Intended Support ? = Unspecified

Does AdaSAGE speak SQL?

Speaking SQL usually refers to a Relational Database Management System (RDBMS). AdaSAGE is not an RDBMS. AdaSAGE is an application development tool that provides facilities for creating an application specific relational database.

There are two aspects of SQL dialog to consider, first, is listening to SQL and responding by executing the requested command; second, is issuing SQL to get a foreign system to execute some process on your behalf. In the first case, AdaSAGE provides both an imbedded SQL technology and an interactive SQL system adapted to comply with ANSI-SQL DML Level 1. In the second case, AdaSAGE does not provide any capabilities for creating SQL commands, but because AdaSAGE is a set of Ada packages there is no reason that a package could not be developed to do so.

Does AdaSAGE support multiuser system development?

AdaSAGE provides for both single user and multiuser systems. Both types of systems have been represented in fielded applications. A unique feature of the AdaSAGE multiuser system (MultSage) is the distribution of not only the data to be shared by many users, but also the distributed processing concept where each user contributes his CPU to the multiuser database management and locking functions. This concept is used for local area network (LAN) type applications.

Are there audit trail and recoverability features in AdaSAGE?

The capability to record all transactions and roll forward from previous dates gives an audit trail and recover capability. These features are often provided within database management systems, and are provided with AdaSAGE as a logging option, but seldom if ever are they used in final applications because of the excessive time and data storage requirements. An application using AdaSAGE may be developed to provide full or partial capabilities in this area using the Ada language and AdaSAGE provided packages. More often, applications regularly backup or copy current data to another device to provide data recoverability. As well, AdaSAGE provides a facility to rebuild or recover current database information in place.

Will AdaSAGE interface to DBMS's?

Neither Codd's rules for a relational system nor ANSI SQL specifies the physical storage format of data. Logically, it must appear to be two dimensional and is often referred to as being composed of relations (or a less precise term, flat files). Each database system, therefore, tends to have its own unique method of physical data storage. A method to move data between different systems is often required. There are some commonly used formats of files from which many DBMS's load and unload data. This is one method of interface between two DBMS's and is supported by AdaSAGE. With AdaSAGE, the Ada language may also be used to directly and much more quickly move data from a DBMS into an AdaSAGE relation or in the reverse direction. The only requirement is that the DBMS support an Ada interface.

Is AdaSAGE a code generator?

Currently, there are no components of the AdaSAGE system that generate code. The application paradigm supported by AdaSAGE results in applications that are data driven. These data are contained in a data dictionary containing information about the relational structure of the application's data, the application's user presentation, and various other application specific information.

Are AdaSAGE packages reusable?

AdaSAGE provides many Ada packages, all of which are reusable. Many of them are independent and may be used within any Ada system without causing the inclusion of the database or user interface facilities. These include libraries for sorting, data movement, binary operations, bit manipulations, graphics, string manipulation, and others. There are also packages that directly access the database system and others that access the user interface facilities.

AdaSAGE adheres to the concept of "black box" modularity, which emphasizes the reuse of code through data encapsulation and extensibility through inheritance.

All of the provided Ada packages are designed to be understandable, useful and usable, and are loosely coupled and highly cohesive at their level of abstraction.

Does AdaSAGE support graphics development?

AdaSAGE graphics support is available on some platforms. The graphics support includes the following:

Low level support
ANSI-GKS Level ma

Intermediate level support
Primitive Calls - Rotatable Arc, PolyLine, Rectangle, Rotatable, Scalable Text and Bit-Mapped Text
Input Calls - Keyboard and Mouse
System Calls
Window Calls
Device Calls
Font Calls
Locator Calls
PCX Graphics File Calls

High level support
2-D Graphing Package - Pie, Bar, X-Y, Area and Line Charts
Graphical User Interface Package - Menu bar, Pull-down Menus and Pop-up Menus
Data driven graphical reports edited in THOR

Device support

Video - CGA, EGA, VGA
Printers - EPSON-FX, EPSON-LQ, HPCL, Postscript
Mouse - Microsoft Mouse Driver

Support Utilities

Resource Compiler
Stroke Font Editor
Bit-Mapped Font Editor
Cursor Editor

This material resulted form work developed under Government Contract No. DE-AC07-76ID01570 at the Idaho National Engineering Laboratory and is subject to "Limited Government License". Neither the United States nor the United States Department of Energy, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights.

Ada Resources Panel

Moderator: Sue Carlson, Ada Information Clearinghouse

Panelists: Deborah Neve, Ada Information Clearinghouse
Representative of Public Ada Library
Representative from SIGAda's Education Working Group
Representative of Ada9X GNAT Team

Ada

The International Language
for Software Engineering

SOURCES FOR Ada INFORMATION AND Ada SOURCE CODE

Susan Carlson
IIT Research Institute
4409 Forbes Boulevard
Lanham, Maryland 20706

ABSTRACT

A variety of resources are available to assist software professionals in developing applications in Ada. The Ada Resources panelists will provide a summary of some of the FREE resources that may be of most interest to ANCOST conference attendees. This paper summarizes the services available FREE from the Ada Information Clearinghouse and sources for FREE reusable Ada source code. For more information about Ada, contact the Ada Information Clearinghouse (AdaIC) at 1/800-232-4211 (voice) or adainfo@ajpo.sei.cmu.edu (Internet).

INTRODUCTION

A variety of resources are available to assist software professionals in developing applications in Ada. The Ada Information Clearinghouse (AdaIC) is an information center devoted to providing pointers to Ada resources. A number of software repositories provide FREE Ada source code. This paper summarizes the products and services available from the AdaIC as well as sources for reusable Ada software.

I. AdaIC PRODUCTS & SERVICES

The latest information about Ada is available to you FREE of charge from the Ada Joint Program Office's (AJPO) Ada Information Clearinghouse (AdaIC). The AdaIC makes available information on a variety of topics -- ranging from the use of Ada within DoD and industry to tools and compilers for Ada developers, and from DoD policies regarding Ada to reusable Ada software.

AdaIC Telephone Hotline: 800/AdaIC-11 or 703/685-1477

For answers to your Ada questions, call the AdaIC, Monday through Friday, 8:00 a.m. to 5:00 p.m., EST.

AdaIC Newsletter

The AdaIC quarterly newsletter provides 18,000 software professionals throughout the world with current news from the AJPO about the Ada program. If you would like to receive the newsletter, call the AdaIC and request a FREE subscription.

Information Flyers/Files

You can obtain more than 100 different information flyers/reports from the AdaIC. Flyer topics include:

- Ada Validated Compilers
- Ada News and Current Events
- Ada Applications
- Ada 9X Project Status
- On-line sources of Ada Source Code
- Ada Education and Training
- Ada Software, Tools, and Interfaces
- Ada Regulations, Policies, and Mandates
- Ada Historical Information
- Standards and Available Ada Bindings Products
- Ada Publications

AdaIC flyers are available electronically on two AJPO-sponsored computer systems: the AJPO host computer on the Internet, and the AdaIC Bulletin Board System (BBS). Additionally, paper copies of the flyers can be ordered by completing the order form on the last page of this article and returning it to the AdaIC.

On-line Information

Most AdaIC flyers and other publications are available on-line on the Internet through the AJPO host computer and on the AdaIC BBS. These electronic sources also have other files, such as those whose length or complexity preclude easy distribution in paper-copy form.

The AJPO host on the Internet
ajpo.sei.cmu.edu
The AdaIC makes a variety of Ada information available in the /public directories on the AJPO host computer on the Internet (AJPO.SEI.CMU.EDU). Some of the most often requested information includes:

public/ada-info
    informational files offered by the AdaIC.

public/ada9x
    information from the Ada 9X Project Office -- including the DRAFT Language Reference Manual for Ada 9X and the DRAFT Rationale for Ada 9X, various other project reports, announcements, and reports from the Ada 9X Project manager.

public/adastyle
    "Ada Quality and Style: Guidelines for Professional Programmers, Version 2.01.01, December 1992". This is the AJPO's suggested Ada style guide for use in DoD programs.

public/asis
    "Ada Semantic Interface Specification (ASIS)", Version 1.1.0, July 1993.

public/atip
    source code that was produced under the AJPO's Ada Technology Insertion Program (ATIP).

public/bindings
    AdaIC report, "Available Ada Bindings" -- in ASCII and Postscript format, and IEEE POSIX/Ada binding package specifications.

public/comp-lang-ada
    Frequently Asked Questions (FAQ) files and archived digests from the comp.lang.ada newsgroup on USENET.

public/newsflash
    weekly summaries of Ada news.

The AJPO host can be accessed via the File Transfer Program (FTP), which allows a user to transfer files to and from a remote network host site. Files on the AJPO host can also be accessed through a mail server, which automatically sends users the files they request through electronic mail.

FTP Access. FTP should work for any host on the Internet.

A sample FTP connection follows.

```
[your-prompt] ftp ajpo.sei.cmu.edu
          execute ftp from your remote site.
name: anonymous
          login using "anonymous"
password: guest
          enter password of "guest"
ftp> cd public
          change to the "public" sub-directory
ftp> dir
          view a list of accessible sub-directories
ftp> cd [ sub-directory]
          change to the sub-directory of your choice
ftp> dir
          view a list of available files
ftp> get file1.hlp [newname.hlp]
          get "file1.hlp" from ftp and copy it to
          newname.hlp" on your machine.
ftp> mget file1..fileN
          get multiple files from ftp and load the
          onto your machine with the same names.
ftp> bye
          logout when finished.
```

For more help information on the AJPO host, type get README" and "get README.FTP" after an ftp connection is made.

Mail Server: ftpmail@ajpo.sei.cmu.edu.
The AJPO's mail server, ftpmail, can be used by anyone with the capability to send and receive electronic mail over the Internet.

To retrieve a specific file, you will need to know the name and directory in which the file resides. To get help, including information on accessing the directories of files, send the following message from your electronic mail account:

To: **ftpmail@ajpo.sei.cmu.edu**
Subject: **help**

AdaIC Bulletin Board System (BBS)
Commercial: 703/614-0215;
Autovon: 224-0215

The source for the latest Ada news is the AdaIC BBS. Each time you log onto the bulletin board, you will have the opportunity to review the week's hottest Ada news stories. In addition, you can download any AdaIC flyer/file or report from the BBS,and you can search AdaIC databases on the BBS.

The BBS can be accessed by dialing one of the numbers listed above. Users should set their telecommunications package with the following parameters: Baud rate = 300 through 9600 baud; Data Bits = 8; Parity = none; Stop Bits = 1. The first time you log on, you will be prompted to register for an account.

For More Information

Ada Information Clearinghouse (AdaIC)
P.O. Box 46593
Washington, DC 20050-6593
Phone: 703/685-1477, 800/AdaIC-11
FAX 703/685-7019
Internet: adainfo@ajpo.sei.cmu.edu
CompuServe: 70312,3303

Ada Applications

Ada's globally documented successes include usage across a wide range of applications. Today Ada drives:

Payroll systems, commercial banking systems, stock quotation transaction systems, language translation system.

Geophysical exploration and data processing systems, and chemical analysis systems.

Commercial cellular phone switching and commercial telecommunications applications.

Commercial jets, air traffic control systems, in-flight detection and guidance systems, flight training simulators, and flight control/flight display systems.

NASAs Space Shuttle and Space Stations.

Automated manufacturing and materials handling systems, robotics welding systems, and inventory management systems.

Real-time continuous medical monitoring systems and real-time embedded control of copier and duplicator products.

Strategic military embedded systems, the majority of which are used in real-time applications, systems and mission trainers.

II. SOURCES FOR REUSABLE Ada SOURCE CODE

The following repositories provide FREE Ada components. Note, some require potential users to apply an register for an account.

Ada Join Program Office (AJPO) Internet Host and AdaIC BBS
Source code from some AJPO-sponsored projects it available to anyone through the AdaIC BBS (703/614-0215, AV 224-0215) and the AJPO host (ajpo.sei.cmu.edu) on the Internet. There is no need to establish an account with the AJPO.

Asset Source for Software Engineering Technology (ASSET)
ASSET is a software reuse library available to software developers in government, industry, and education. ASSET components can be generally categorized as: Ada Standards and Bindings, Data Base Management Systems, Software Development Process, Software Engineering Environment, and Software Reuse Technologies. Users interested in applying for an ASSET account should contact: ASSET Accounts, 2611 Cranberry Square, Bldg. 2600, Suite 2, Morgantown, WV 26505; Telephone: (304) 594-1762; FAX: (304) 594-3951; E-mail: info@source.asset.com

Computer Software Management and Information Center (COSMIC)
COSMIC Distributes software developed by NASA, including a number of Ada programs. Source code and documentation can be purchased separately, and abstracts are available free of charge. For more information, contact: COSMIC, The University of Georgia, 382 East Broad Street, Athens, GA 30602, 404/542-3265.

Defense Software Repository System (DSRS)
DSRS is an automated library of reusable software development components available to the DoD and other Government agencies, including supporting contractors. Users must register to use the DSRS. For more information on the DSRS, contact: DoD Center for Software Reuse Operations, Customer Assistance Office, 500 North Washington Street, Suite 101, Falls Church, VA 22046; 703/536-7485; Fax: 703/536-5640.

National Technical Information Service (NTIS)
NTIS is a self-supporting publishing agency for the U.S. Department of Commerce. It provides a free catalog of the software available from the Federal Computer Products Center, which is a clearinghouse for over 3500 products from about 100 Federal agencies. Contact: NTIS, U.S. Dept. of Commerce, 5825 Port Royal Road, Springfield, VA 22161, or call the nonprint product manager (Federal Computer Products Center), 703/487-4650, or 800/553-NTIS.

Public Ada Library (Ada Software Repository)
The Public Ada Library (PAL) is a collection of Ada programs, tools, and educational materials. Anyone with Internet access may retrieve source code via ftp (wuarchive.wustl.edu). The PAL is available for sale on diskette, CD-ROM, and tape from a number of sources.

AdaIC Flyers

Ada News, Publications, & Organizations

- ❐ G14 Ada Calendar — events such as national and international conferences on specific applications and technical challenges.
- ❐ G22 Ada Today — announcements, reports on products, events, etc.
- ❐ G21 AdaIC Newsletter — current issue.
- ❐ S59 AdaIC newsletters, past year — four issues.
- ❐ G46 SIGAda membership form.
- ❐ G49 Ada Serial Publications — magazines, trade journals, etc.

History

- ❐ G31 "The History of Ada" — magazine article.
- ❐ S68 Ada Trademark Replaced by Certification Mark — 1987 AJPO announcement.
- ❐ S50 Background on ISO's Acceptance of Ada as an international standard — 1987.
- ❐ S19 The Naming of Ada, and "The Real Ada: Countess of Lovelace" — biographical article.
- ❐ S55 NATO Adoption of Ada in Military Systems — 1985.

Policy

- ❐ S78 USAF Interpretation of FY 1991 DoD Appropriations Act — Congresssional Ada Mandate.
- ❐ S98 Air Force Ada Implementation Plan — 1989.
- ❐ S17 Air Force Policy on Programming Languages — 1990.
- ❐ S107 Army Ada Implementation Plan — 1992.
- ❐ S109 ASD (C3I) Waiver-Guidance Memorandum — 1992.
- ❐ S34 The Common APSE Interface Set (CAIS-A) — how to obtain.
- ❐ G36 The Congressional Ada Mandate - 1990.
- ❐ S25 DoD Directive 3405.1, Computer Programming Language Policy.
- ❐ S96 How to Obtain Navy's Ada Implementation Guide — 1992.
- ❐ S35 Marine Corps Ada Implementation Plan — 1988.
- ❐ S41 Software Executive Officials — current list.

On-line Sources of Ada Information

- ❐ G40 AdaIC Products and Services.
- ❐ S66 AdaNET Executive Summary — describes on-line service.
- ❐ G48 Access to Ada Information on the Internet.
- ❐ S110 Asset Source for Software Engineering Technology (ASSET).
- ❐ G102 Defense Software Repository System.
- ❐ S02 Electronic Mail to DDN through Public Access Sites — for those without an account on the Defense Data Network.
- ❐ S26 How to Acquire Files from the Ada Software Repository.
- ❐ G71 AdaIC Databases Available On-Line: Ada Products & Tools, Ada Pragma Support, Ada Article Abstracts, Ada Bibliographies — descriptions.
- ❐ G51 Public access to the Ada Information Clearinghouse Bulletin Board — how to access the AdaIC bulletin board at 703/614-0215.
- ❐ G20 Ada Source Code, Reusable Components, and Software Repositories —products and sources of code.

Ada Applications

- ❐ G89 "10 Ada Successes" — reprint of news article.
- ❐ S84 Ada and Embedded Systems — overview article.
- ❐ S08 Ada Usage Database (summary list — no contact names).
- ❐ G29 Ada Usage Database survey form — for projects.
- ❐ G108 Commercial Applications in Ada.
- ❐ S44 "Why Ada is not Just Another Programming Language" — 1986.

Ada Compilers and Tools

- ❐ S77 Ada Design Language/CASE Developers Matrix
- ❐ V09 Ada Compiler Validation Capability Implementers Guide — how to obtain.
- ❐ V04 How to Obtain Latest ACVC Test Suite — validation test suite.
- ❐ V07 How to Obtain the Ada Compiler Evaluation Capability (ACEC) — the DoD's compiler-performance test package.
- ❐ G28 How to Obtain the Ada Language Reference Manual.
- ❐ V42 Ada Validation Facilities — current list.
- ❐ S93 Ada bindings-products survey form — for vendors.
- ❐ S92 Products and Tools Database survey form — for vendors.
- ❐ S87 Services Embrace AdaSAGE — discussion of government-domain Ada application-development system.
- ❐ S82 Available Ada Bindings — description of standards, with list of available products.
- ❐ V15 How to Obtain Benchmark Performance Test Suites and Results.
- ❐ G103 Free Ada interpreter available for downloading — Ada/ED for Ada 83.
- ❐ G111 Free Source code for GNAT Compiler — available on Internet.
- ❐ G85 How to Obtain *Ada Quality and Style: Guidelines for Professional Programmers*, Version 2.0, 1991.
- ❐ G94 Life of ACVC 1.11 Validation Suite Extended — text of April 14, 1992, AJPO announcement.
- ❐ G10 Validated Compilers List — current list.
- ❐ S47 Ada Compiler Validation Procedures Available — how to obtain.
- ❐ V60 Validation Summary Reports — results from compiler-validation tests, and ordering info from NTIS/DTIC.
- ❐ V58 Withdrawn Tests List — withdrawn from validation test suite ACVC 1.11.

Education and Training

- ❐ G18 Ada Books — books published from 1981.
- ❐ S39 Ada Training Videotapes Available Through National Audiovisual Center — description and ordering info for AJPO-sponsored series.
- ❐ S52 Classes & Seminars — scheduled and self-paced courses nationwide.
- ❐ S83 CREASE Survey Form — for those giving classes.
- ❐ S11 Catalog of Resources for Education in Ada and Software Education — how to obtain.
- ❐ S112 Proposals Sought for Curriculum Development in Software Engineering and Ada.

Ada 9X Project

- ❐ S100 "Contrasts: Ada 9X and C++".
- ❐ S30 Ada 9X Update — current report to the public.
- ❐ G75 Overview of "Ada and C++: A Business Case Analysis" — executive summary of Air Force study.
- ❐ S101 An Overview of Ada 9X — a programmer's perspective.
- ❐ S104 How to Program in Ada 9X, Using Ada 83 — guidelines for compatibility, by Erhard Ploedereder.
- ❐ S105 Multiple Inheritance in Ada 9X — Language Study Note by S. Tucker Taft.
- ❐ S95 The Ada 9X Project: An Overview — a general perpective.
- ❐ S114 Compatibility between Ada 83 and Ada 9X, by John Barnes

NAME (TITLE OPTIONAL)

COMPANY NAME

ADDRESS

PHONE FAX

E-MAIL

Ada

The International Language
for Software Engineering

Ada IC

Ada Information Clearinghouse
sponsored by the
Ada Joint Program Office

FLYER ORDER FORM

Ada Information Clearinghouse publications are available free of charge. Just check off which forms you need, write your name in the space provided, and return the form to us.

For your convenience, most of these flyers are available for downloading from the AdaIC Bulletin Board (703/614-0215 or A/V 224-0215) and the AJPO Host on the Internet (ajpo.sei.cmu.edu). For more information, order flyers numbered G48 and G51 under "On-line Sources of Ada Information."

Ada IC

Ada Information Clearinghouse
P.O. Box 46593
Washington, D.C. 20050-6593

[fold over front cover and tape here]

Author's Index

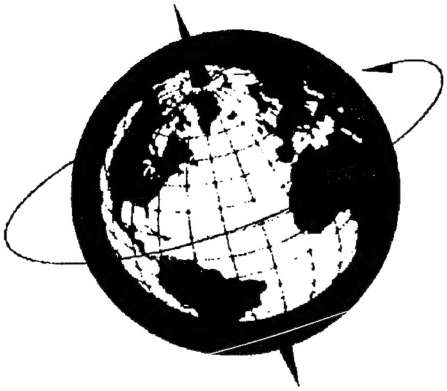
Page

Adams, Sandy H 84
Agrawal, Jagdish 97, 124
Ahmed, Aftab 51
Al-Dhelaan, Abdullah 97, 124
Al-Haddad, Hisham 89, 184
Barber, Gary H. 195
Blundell, James K. 162
Brashear, Phil 29
Briggs, Robert O. 258
Bruan, Christine 77
Chu, Huei-Chung 134
Chung, Ke-Hsiung 134
Clark, John 69
Coleman, Don M. 6
Dai, H. K. 129
Donald, John 69
Fay, Bill 16, 36
Flannery, K. E. 129
Gotterbarn, Donald 167
Graziano, Amy P. 108
Harrison, George 148
Hooper, James W. 184
Jenkins, Mike 20
Kelsh, James P. 154
Leach, Ronald J. 6
Lee, Roger Y. 154
Lodgher, Akhtar 89, 184
McArthur, G. William 142
Meyer, Charles A. 195
Monroe, Joseph 116, 248
Nunamaker, Jay 258
Owens, Ronald L. 226
Potoczniak, Joseph J. 206
Pukite, Paul R. 63
Riggs, Marrea H. 226
Riser, Robert 167
Ruth, Rick E. 142
Scandura, Joseph M. 103
Schoen, William 240
Shetler, Toni 216
Smith, Suzanne 167
Soto, David 195
Stevens, Barry 69
Stukes, Sherry 69
Wu, C. Thomas 252
Yu, Huiming 116, 248
Yuan, Bau-Hong 134

Panel's Index

Page

Dual Use Panel 288
Ada9x Panel 293
Environment Assessment Methodology Panel 298
HBCU Net Panel 300
DOD Open Systems Environment Guidance Panel 307
AdaSAGE 323
Ada Resources Panel 332



Ada

The International Language
for Software Engineering